Yksikkö- ja integrointitestauksen menetelmät ja mittarit

Anita Satukangas

Helsinki 22. toukokuuta 2003

Pro gradu -tutkielma

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Sisällys

1 Johdanto 1

2 Yleistä testauksesta 2

2.1 Mitä on testaus 2

2.1.1 Testausmenetelmät 4

Lasilaatikkotestaus 5
Mustalaatikkotestaus 6

2.2 Testauksen V-malli 7

2.2.1 Yksikkötestaus 9

2.2.2 Integrointitestaus 10

2.2.3 Järjestelmätestaus 12

2.2.4 Hyväksymistestaus 12

3 Virheiden luokittelu 13

3.1 Vika – virhe – häiriö 13

3.2 Perusteet virheiden luokittelulle 14

3.3 IEEE-standardi 1044 ohjelmistoanomalioiden luokittelusta 15

3.3.1 Luokittelun vaiheet 15

3.3.2 IEEE1044-luokittelun käyttöönotto 19

3.3.3 Luokitusten hyödyntäminen 21

3.3.4 Valmiin luokittelun tarjoamat edut 21

3.4 Ortogonaalinen virheluokittelu 22

3.4.1 Aktiviteetti 24

3.4.2 Virheen tyyppi 24

3.4.3 Virheen laukaisin 26

3.4.4 ODC-luokittelun käyttöönotto 28

4 Menetelmät 30

4.1 Testaustehtävät ja -menetelmät 30

4.2 Testauksen dokumentointi 36

5 Yksikkö- ja integrointitestauksen mittarit 39

5.1 Rakenteellisen testauksen mittarit 39

5.1.1 Polkutestaus 39

5.1.2 Tietovuotestaus 43

5.1.3 Mutaatiotestaus 44

5.1.4 Muita rakenteellisen testauksen mittareita 44

5.1.5 Kattavuuden suhde ohjelmakoodin laatuun 44

5.1.6 Polkutestauksen rajoitukset 46

5.2 Toiminnallisen testauksen mittarit 46

5.2.1 Virheiden lukumäärä mittarina 46

5.2.2 Mikä on virhe 47

5.2.3 Virhetiheys 48

5.2.4 Virheiden priorisointi testauksessa 51

5.3 Prosessimittarit 53

5.3.1 Virheenpoiston tehokkuus 53

5.4 Mittareiden käyttö 55

6 Yhteenveto 56

Lähteet 58

Kuvat

Kuva 1 Testauksen V-malli 8
Kuva 2 IEEE1044-standardin mukainen ohjelmistopoikkeaman luokitteluprosessi sekä eri vaiheiden pakolliset ja täydentävät luokittelutiedot..... 16
Kuva 3 Esimerkki IEEE1044-standardin tyyppiluokitteluhierarkiasta 18
Kuva 4 Yksinkertaistettu malli virhetyyppien jakaumassa tapahtuvasta muutoksesta sovelluskehitysprosessin eri vaiheissa [CBC92]..... 25
Kuva 5 DB2-tietokantatatuotteen laukaisinten jakauma tuotantokäytöstä kolmen vuoden ajalta [CBC92]..... 27
Kuva 6 IEEE1008 standardin määrittelemät yksikkötestauksen vaiheet ja toimenpiteet..... 31
Kuva 7 Havaitun virheen käsittely IEEE-standardin 1008 mukaan 33
Kuva 8 Lohkokaavio. Ehto-/predikaattisolmu p sekä suoritettavat lauseet s1 ja s2, nuolet kaaria 40
Kuva 9 Lohkokaavioesimerkki 41
Kuva 10 Kattavuushierarkia 43
Kuva 11 Ohjelmakoodista löydettyjen virheiden (=laatu) suhde kattavuuteen. Laatuasteikko kertoo, kuinka suuri osa ohjelman kaikista virheistä on löydetty [WMM01]. 45
Kuva 12 Virheiden syntyyn ja löytymiseen vaikuttavia tekijöitä [FeN00] 47

Taulukot

Taulukko 1 Virheiden asiakasvaikutus-ominaisuuden luokittelu ODC:n versiossa 5.1 [BKS98] 23
Taulukko 2 Virheen korjauksen yhteydessä tunnistettavat ominaisuudet ODC-luokittelun versiossa 5.1 [BKS98] 23
Taulukko 3 Eri virhetyyppien tavanomaisimmat synty- ja löytöpaikat [CBC92]..... 26
Taulukko 4 Ohjelmakehityksen eri vaiheissa (aktiviteetti) ilmenevät virheen laukaisimet ODC-luokittelun versiossa 5.1 [BKS98] 27
Taulukko 5 Verifioitavan tai validoitavan yksikön kriittisyys [IEEE1012]..... 34

1 Johdanto

Ohjelmistotuotannon kustannukset ovat huomattavia. Ohjelman koko elinkaaren aikaisista kustannuksista toiseksi kallein on testaus, joka muodostaa kaikista sovelluksen elinkaaren kuluista yhteensä noin 45 %[1] [MaM83]. Testauksen tehostamisella voitaisiin ajatella saavutettavan merkittäviäkin kustannussäästöjä.

Ohjelmaa testataan ja siitä löytyneitä virheitä korjataan useissa vaiheissa ohjelman elinkaarta. Virheiden löytäminen ja korjaaminen pienistä yksiköistä on helpompaa kuin suuremmista. Sovelluskehityksen edistyessä myös toiminnallisuus monimutkaistuu eri komponenttien, käyttöjärjestelmien ja laitteistojen välisen yhteistoiminnan seurauksena. Virheiden korjauksen aiheuttamien kustannusten katsotaankin yleensä nousevan eksponentiaalisesti sen myötä, mitä myöhäisemmässä vaiheessa ohjelman elinkaarta ne löydetään [KFN99]. Toisaalta viime aikoina on myös ilmennyt eriäviä näkökantoja, joiden mukaan ainoastaan vakavien virheiden korjaus olisi tuotantovaiheessa olevan ohjelman osalta 100 kertaa kalliimpaa kuin vaatimus- ja suunnitteluvaiheessa ja tavanomaisten virheiden osalta ero olisi vain on yhden suhde kahteen [SBB02]. Virheiden löytäminen ja korjaus on kuitenkin aina kalliimpaa myöhäisemmissä ohjelmistotuotantoprosessin vaiheissa ja siksi kustannussäästöjä on saavutettavissa panostamalla virheiden löytymiseen mahdollisimman aikaisessa vaiheessa.

Tutkielmani tarkoituksena on selvitellä niitä menetelmiä, joilla aidossa liiketoimintaympäristössä toimiva ja kilpaileva ohjelmistotuotantoyritys voi tehostaa testauksen varhaisvaiheita, yksikkö- ja integrointitestausta, ja minkälaisin mittarein näitä testausvaiheita on mahdollista mitata. Tarkoitukseni on nostaa esiin sellaisia menetelmiä ja mittareita, joita voidaan käyttää testausprosessin varhaisvaiheiden kehittämisen alkuvaiheessa ikään kuin ensimmäisinä askeleina kohti hallitumpaa testausprosessia. Rajaan tutkielmani tarkastelukulman koskemaan ei-turvallisuuskriittisiä järjestelmiä tuottavia yrityksiä. Tällöin yrityksen toiminnan keskeiseksi tavoitteeksi

[1] Kallein yksittäinen vaihe on tuotteen ylläpito, joka Martin & McCluren [MaM83] muodostaa noin 67 % kaikista kuluista. Testauksen osuus ylläpitovaiheen kuluista on noin 30 %.

muodostuu kannattava liiketoiminta, jonka perusteella myös testauksen menetelmien ja mittareiden aiheuttamia kustannuksia ja tuottamia hyötyjä arvioidaan.

Testausautomaation olen tarkoituksellisesti rajannut tutkielman ulkopuolelle, sillä testauksen automatisointi tulee ajankohtaiseksi vasta, kun testauksen toimintatavat ja menettelyt on määritelty ja ne ovat aidosti integroituneet osaksi ohjelman tuotantoprosessia. Vasta tässä vaiheessa automaatiolla voidaan saavuttaa kustannussäästöjä, kun huomioidaan automatisoinnin tuomat suuret kustannukset laitehankintoina, koulutuksena, toimintatapojen muutoksena sekä alkuvaiheen alentuneena tuottavuutena. Kuten Kaner, Bach & Pettichord asian ilmaisevat: "Älä automatisoi sotkua" [KBP02]. Käytännön soveltamisessa tarvittavan testausautomatiikan vuoksi myös rakenteellisen testauksen mittarit on tutkielmassa käsitelty vain hyvin yleisluonteisesti.

Tutkielman rakenne on seuraava: luvussa 2 käydään läpi käsitteen testaus sisällössä aikojen kulussa tapahtuneet muutokset, keskeiset testausmenetelmät lasi- ja mustalaatikkotestaus sekä testauksen eri vaiheet ja niiden kytkentä konstruktiivisiin ohjelmistotuotantoprosessin vaiheisiin niin sanotun V-mallin avulla. Luku 3 esittelee kaksi virheiden luokittelumallia: IEEE:n standardin 1044 ja ortogonaalisen virheluokittelun sekä niiden soveltamisen ja käyttöönoton yrityksessä. Luvussa 4 määritellään yksikkö- ja integrointitestauksen menetelmät testaustehtävineen sekä näissä vaiheissa toteutettava dokumentointi. Luvussa 5 käydään läpi testauksen varhaisvaiheissa mahdollisia mittareita, niiden soveltamista ja riskejä. Luku 6 on yhteenveto.

2 Yleistä testauksesta

2.1 Mitä on testaus

Nykykirjallisuudessa testauksen määrittely noudattelee useimmiten Meyersin vuonna 1979 julkaistua teosta The Art of Software Testing, jossa testaus määritellään ”prosessiksi, jonka tarkoituksena on ohjelman suorittaminen virheiden löytämiseksi” [Mey79]. Testauksen keskeinen tavoite on siis löytää ohjelmasta virheitä [Pre00, Bei90].

Testauksen tavoitteet sekä se, mitä sovelluskehitysprosessin aktiviteetteja termin alle katsotaan kuuluvaksi, ovat vaihdelleet aikojen kuluessa. Gelperin ja Hetzel kuvaavat artikkelissaan The growth of software testing [GeH88] testauksen neljä historiallista kehitysvaiheitta. Digitaalisen tietojenkäsittelyn alussa testaus keskittyi laitteistoon. Ohjelmistojen osalta manuaalista tarkistamista, virheenjäljitystä (debug) ja testausta ei juurikaan eroteltu toisistaan. Tätä kautta he kutsuvat virheenjäljityssuuntautuneeksi aikakaudeksi (Debugging-Oriented Period).

Ohjelmien määrän lisääntymisen sekä niiden kompleksisuuden ja tuotantokustannusten kasvun myötä ohjelmistotuotantoon liittyvä taloudellinen riski kasvoi, mikä puolestaan lisäsi testauksen merkitystä. Tutkijoiden mukaan testauksen osalta siirryttiinkin 1950-luvun lopulla toteennäyttösuuntautuneeseen jaksoon (Demonstration-Oriented Period). Virheenjäljitys erotettiin varsinaisesta testauksesta, vaikkakin molemmat käsitteet edelleen pitivät sisällään virheiden etsimistä, tunnistusta, paikannusta ja korjausta. Keskeinen ero oli tavoitteessa. Virheenjäljityksellä varmistettiin, että ohjelma ylipäätään toimi, kun taas testauksen keskeinen tavoite oli osoittaa, että ohjelma täytti sille asetetut määrittelyt.

Vuonna 1979, Meyersin klassikkoteoksen ilmestymisen myötä, siirryttiin tutkijoiden mukaan hävittämissuuntautuneeseen aikakauteen (Destruction-Oriented Period). Testauksen tavoitteeksi nousi ohjelman virheiden löytäminen. Tässä vaiheessa testaus myös kytkeytyi voimakkaasti muihin virheiden etsimistoimenpiteisiin kuten ohjelman analysointiin ja katselmointiin.

Tutkijoiden mukaan 1980-luvun alkupuolella alkoi arviointisuuntautunut aikakausi (Evaluation-Oriented Period). Siinä testaus nähdään osana metodologiaa, jossa integroidaan ohjelman analysointi, katselmointi ja testaustoimenpiteet ja jonka pyrkimyksenä on tuotteen arviointi sen koko elinkaaren ajan. Tausta-ajatuksena on:

> Mikään yksittäinen verifiointi, validointi tai testaustekniikka ei voi taata oikein toimivaa, virheetöntä ohjelmaa. Kuitenkin huolella projektia varten valittu joukko näitä tekniikoita voi auttaa varmistamaan, että projekti tuottaa ja ylläpitää laadukkaita ohjelmia [Gui83].

Viimeisimpänä kehityssuuntana tutkijat pitävät estosuuntautunutta aikakautta (Prevention-Oriented Period), joka heidän mukaansa alkoi 1980-luvun lopulla. Estosuuntautuneessa näkemyksessä testauksen keskeinen tavoite on virheiden estäminen. Tavoite saavutetaan kytkemällä testaus mukaan sovelluskehitysprosessiin jo sen alkaessa. Testiaineiston suunnittelu prosessin aikaisessa vaiheessa vaatimusten ja arkkitehtuurimäärittelyjen perusteella aiheuttaa kysymyksiä, jotka voivat paljastaa lähdeaineistosta riittämättömyyttä, ristiriitaisuutta, tulkinnanvaraisuutta ja suoranaisia virheitä silloin, kun niiden korjaaminen vielä on yksinkertaista ja kustannuksiltaan vähäistä verrattuna myöhäisempiin kehitysvaiheisiin. Testauksen huomioiminen siis parantaa määrittelyjen ja ohjelmakoodin laatua jo ennen varsinaisten testitapausten suorittamista. Testausprosessi nähdään eräänä riskienhallinnan välineenä, ja testausstrategia ja testitapausten suunnittelu perustetaan ohjelman suorituksessa ilmenevien häiriöiden aiheuttamien seurausten ja näiden häiriöiden todennäköisyyden arvioinnille.

Gelperin ja Hetzel näkemys testauksesta ja sen roolin muutoksista tietojenkäsittelyn lyhyen historian kuluessa on mielestäni varsin osuva. Eri kaudet eivät tietenkään ole todellisuudessa olleet näin selvärajaisia, vaan heidän määrittelemiensä kausien aikana on rinnan elänyt useita tutkijoiden luokituksessa eri kausille kuuluvia testauskäsityksiä. Kilpailu ja kustannustekijät painottavat yhä enenevässä määrin erilaisia tekniikoita ja menetelmiä, joilla pystyttäisiin sekä estämään virheiden syntyä ohjelman elinkaaren aikana että eliminoimaan jo syntyneet mahdollisimman aikaisessa vaiheessa.

2.1.1 Testausmenetelmät

Testausmenetelmät voidaan karkeasti jakaa kahteen luokkaan: rakenteelliseen testaukseen (structural testing), jossa ohjelman sisäiset rakenteet ja yksityiskohdat ovat testaajan nähtävissä ja hyödynnettävissä testitapausten suunnittelussa, sekä toiminnalliseen testaukseen (functional testing), jossa testattavaa ohjelmaa tai järjestelmää kohdellaan ikää kuin mustana laatikkona, jonka sisäisestä rakenteesta ja yksityiskohdista testaajalla ei ole tietoa [Bei90].

Lasilaatikkotestaus

Rakenteellisessa eli niin sanotussa lasilaatikkotestauksessa (white box -testing) testitapaukset luodaan tarkastelemalla ja analysoimalla ohjelman lähdekoodia [Bin00]. Lasilaatikkotestauksessa ohjelmaa voidaan testata paloina, jolloin testaus voidaan paremmin keskittää ongelmallisten moduulien läpikäymiseen hyvin tarkalla tasolla. Testaaja myös näkee, mitkä ohjelman koodirivit, haaraumat tai suorituspolut on testattu. Hän voi tarvittaessa lisätä testitapauksia, joilla saavutetaan sellaisia ohjelmakoodin osia, joita ei vielä ole testattu [KFN99]. Testauksen riittävyyttä ilmaistaan usein kattavuudella (coverage). Se on prosentuaalinen luku, joka ilmaisee kuinka suuren osan valitun kattavuusstrategian perusteella laskettuna kyseisten testitapausten joukko suorittaa koko ohjelmakoodista [Bin00].

Ohjelmakoodin käyttö testitapausten laadinnassa auttaa eliminoimaan testitapauksia, joita mahdollisista toiminnallisuuden eroista huolimatta todellisuudessa käsitellään samoilla rutiineilla ja loogisilla suorituspoluilla [Bei90]. Lähdekoodin tarkastelu mahdollistaa myös ohjelmassa olevia poikkeuskäsittelyjä testaavien testitapausten luonnin [Bin00]. Ohjelman sisäiset raja-arvot, joista muuten on hankala saada tietoa, ovat heti nähtävillä ja niiden testaamiseen voidaan luoda sopivat testitapaukset [KFN99]. Myös ohjelmakoodissa käytetyt algoritmit ovat nähtävillä. Tunnetuimmilla algoritmeilla on useita tiedettyjä virhelähteitä, esimerkkinä matriisin kääntäminen, joka voidaan suorittaa monella tapaa ja johon sisältyy useita hyvin tunnettuja mahdollisuuksia laskea tulos väärin. Nämä voidaan eliminoida lasilaatikkotestauksessa [KFN99]. Rakenteellisen testauksen etuna on myös sen soveltuvuus mustalaatikkomenetelmiä helpommin automatisoitavaksi [DRM96].

Lasilaatikkomenetelmien ongelmana on muun muassa se, että testitapausten laatijalta vaaditaan koodin analysointitaitoja sekä tietoa sovellusspesifeistä implementointitekniikoista [Bin00]. Ohjelmakoodin rakenteen tarkasteluun perustuvia menetelmiä käytetään tyypillisesti vain testausprosessin varhaisvaiheissa; yksikkötestauksesta valtaosa on lasilaatikkotestausta, integrointitestauksesta noin puolet [Pre00]. Juuri tarvittavan osaamisen ja koodituntemuksen vuoksi resursoinnin kannalta on järkevää, että ohjelman tekijä suorittaa ainakin yksikkötestauksen ja käytännössä usein myös integrointitestauksen.

Lasilaatikkomenetelmin ei voida tuottaa tehokasta testaussuunnitelmaa suurten alijärjestelmien tai sovellusten testaamiseen. Ohjelmakoodiin perustuvin menetelmin voidaan ainoastaan todentaa, että ohjelmakoodi tuottaa oikean lopputuloksen tietyllä suorituspolulla. Siihen, vastaako suorituspolku toiminnallisesti sitä, mitä ohjelmalta odotetaan, ei näillä menetelmillä oteta kantaa. Myöskään korkeat kattavuusarvot eivät kerro virheiden puuttumisesta tai ohjelmalle asetettujen vaatimusten täyttymisestä [Bin00]. Mahdollinen toiminnallisuuden puuttuminen tai puuttuvat ohjelman osat eivät myöskään ilmene testattaessa lasilaatikkomenetelmin [Bin00, KFN99]. Rakenteellinen testaus on myös huono menetelmä etsittäessä ajastukseen liittyviä virheitä, odottamattomia virhetilanteita, käyttöliittymän ristiriitaisuuksia, näytöllä näkyvän datan epävalidiutta, taustatoimintojen välistä yhteistoimintaa sekä erilaisia määrä-, kuormitus- ja laitteistovirheitä [KFN99].

Mustalaatikkotestaus

Toiminnallisessa testauksessa (black box -testing) testattavaa ohjelmaa tai järjestelmää kohdellaan ikään kuin mustana laatikkona, jonka sisäisestä rakenteesta tai yksityiskohdista testaajalla ei ole tietoa [Bei90]. Ohjelmalle tai järjestelmälle annetaan syötteitä (input) ja näiden syötteiden käyttäjälle näkyviä tuloksia (output) verrataan oraakkelin antamaan tulokseen oikeellisuuden varmistamiseksi. Oraakkeli on väline tai keino, joka antaa tiedon komponentin (oikeasta) odotetusta käyttäytymisestä [How86]. Toiminnallinen testaus keskittyy siis ohjelman ulkoisesti havaittavan toiminnallisuuden ja piirteiden testaamiseen. Tavanomaisessa kielenkäytössä termillä testaus viitataan juuri toiminnallisuuteen perustuviin menetelmiin.

Ohjelman täydellinen testaaminen, eli kaikkien mahdollisten syötteiden anto ja tulosten verifiointi, on mahdollisten syötteiden valtavan määrän johdosta käytännössä mahdotonta [Bei84, Bin00]. Taloudellisesti järkevän testauksen on siis jollakin menetelmällä valittava ne syötteet, joilla testaus suoritetaan. Yleisimmin syöteavaruus (kaikkien mahdollisten syötteiden arvot) jaetaan osajoukkoihin, joita hyödynnetään testattavien arvojen valinnassa. Tällaisia menetelmiä ovat muun muassa ekvivalenssiluokittelu (equivalence partitioning) sekä arvoaluetestaus (domain testing), jotka molemmat perustuvat mahdollisten syötteiden jakoon osajoukkoihin, joiden alkioiden oletetaan käyttäytyvän ohjelmassa samalla tavoin. Tällöin minkä tahansa osajou-

kon jäsenen testaaminen vastaa koko osajoukon kaikkien jäsenten testaamista [Bin00]. Näin koko syöteavaruus voidaan testata kohtuullisella testitapausten määrällä. Käytännössä on kuitenkin mahdotonta tietää, käsitteleekö ohjelma samaan luokkaan asetut arvot samalla tavoin. Tämän johdosta luokittelu on aina testaajan kokemukseen ja näkemykseen perustuva oletus ohjelman käyttäytymisestä erilaisilla syötteillä, ja sitä voidaan testauksen aikana joutua muuttamaan tai tarkistamaan.

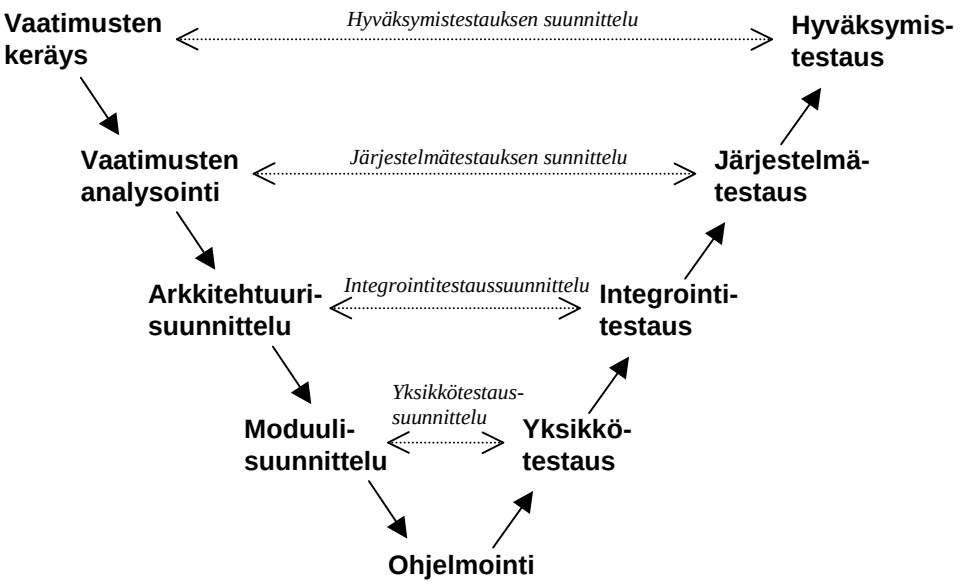
Toiminnallisen testauksen etuna on sen soveltuvuus kaikille testauksen tasoille. Toisin kuin rakenteellinen testaus, mustalaatikkomenetelmät löytävät myös ohjelmasta puuttuvan toiminnallisuuden ja piirteet sekä virheelliset suorituspolut [DRM96]. Mustalaatikkomenetelmien ongelmana puolestaan on, että testauksen teho riippuu suurelta osin sovelluksen määrittelyjen tasosta sekä testitapausten suunnittelijan kyvystä tulkita niitä oikein [McS01].

Toiminnalliseen testaukseen sisältyvät myös muun muassa kuormituksen, suorituskyvyn ja käyttöliittymän testaus [KFN99]. Testausvaiheista järjestelmätestaus ja hyväksymistestaus suoritetaan käytännössä kokonaan toiminnallisin menetelmin.

On tärkeää huomata, että musta- ja lasilaatikkotestaus eivät ole toisiaan poissulkevia, vaan toisiaan täydentäviä lähestymistapoja. Kummallakin on omat soveltamisalueensa. Ne löytävät ohjelmasta tai järjestelmästä erilaisia virheitä, mutta kummassakin on myös omat rajoitteensa [Bei90]. Näiden tekniikoiden väliin sijoittuu joukko erilaisia hybriditekniikoita (harmaalaatikkomenetelmät/gray box testing), joissa vaihtelevassa määrin hyödynnetään molempia testaustekniikoita ja jotka ovat käyttökelpoisia kaikilla testauksen tasoilla [Bei95].

2.2 Testauksen V-malli

Testaus jaetaan yleensä hierarkkisiin tasoihin, jotka kytketään konstruktiivisiin ohjelmistotuotantoprosessin osiin niin sanotun V-mallin avulla [CrJ02, JoE94]. Tavallisesti käytetään jakoa neljään testaustasoon: yksikkö-, integrointi-, järjestelmä- ja hyväksymistestaus (Kuva 1).



Kuva 1 Testauksen V-malli

V-malli kuvastaa testausprosessin eri vaiheiden keskittymistä tarkastelemaan ja verifioimaan tuotettua sovellusta tai sen osia hieman eri perspektiiveistä ja eri abstraktiotasoilta. Testattavan kokonaisuuden koko kasvaa edettäessä testausprosessihierarkiassa ylöspäin. Yksikkötestaus keskittyy testaamaan pienimpiä toiminnallisia yksiköitä. Integrointitestaus puolestaan testaa yksiköiden välistä yhteistoimintaa. Järjestelmä- ja hyväksymistestauksessa kohteena on koko järjestelmän toiminnallisuus. Perusteena testauksen jaolle eri vaiheisiin on siitä aiheutuva työn parempi organisointi ja tästä aiheutuvat kustannussäästöt, sillä virheet on helpompi löytää ja korjata pienistä yksiköistä. Kun ohjelman perusyksiköt on testattu huolellisesti, ylemmät testaustasot kyetään suorittamaan nopeammin ja systemaattisemmin.

V-malli liittää testauksen kiinteästi osaksi sovelluksen kehitysprosessia. Kunkin testausvaiheen suunnittelu tulisikin aloittaa samanaikaisesti V-mallin osoittaman kehitysvaiheen kanssa. Testaussuunnitelman laatiminen vastaavassa kehitysvaiheessa auttaa selventämään ja täsmentämään monia vielä kehitysvaiheessa kenties epäselviä tai epätarkkoja seikkoja ja yksityiskohtia, mikä estää virheiden syntymistä [EiR96, Bei90]. Tämä on eräs keskeinen testauksen tavoite. Virheet ovat halvimpia korjata

juuri suunnitteluvaiheessa ennen kuin ne ovat aidosti "syntyneet" varsinaiseen ohjelmakoodiin.

Kirjallisuudessa ja erilaisissa kaupallisissa testausta käsittelevissä aineistoissa esiintyy myös nelitasoisesta poikkeavia V-malleja. Olennaista V-mallissa ei kuitenkaan ole käytettyjen tasojen määrä, vaan ajatus testausprosessin kytkemisestä varsinaiseen kehitysprosessiin. Tämän lähestymistavan etuina ovat virheiden väheneminen ja testausprosessin kokonaissuoritusajan lyheneminen virheiden löytyessä lähempänä niiden syntypaikkaa ja -aikaa. Kunkin ohjelmistotuotantoyrityksen tuleekin määritellä omien lähtökohtiensa ja ohjelmistotuotantoprosessinsa perusteella testausprosessissa käytettävät testaustasot, niiden määrä, testauskohde ja testaustavoitteet.

2.2.1 Yksikkötestaus

Yksikkötestauksessa ohjelman pieni riippumaton osa (moduuli, alijärjestelmä, luokka, klusteri tms.) testataan lähdekoodin tasolla. Yleensä tällainen yksikkö toteuttaa jonkin pienen loogisen kokonaisuuden. Koska nämä yksiköt tyypillisesti eivät ole itsenäisesti toimivia, niiden testaamiseen tarvitaan erillisiä testiajureita (driver) edustamaan yksikköä kutsuvaa ohjelmaa ja tynkiä (stub) korvaamaan niitä yksiköitä, joita testattava yksikkö kutsuu. Yksikkötestaus toimii perustana seuraaville testausvaiheille ja tulisi siksi suorittaa erityisen huolellisesti.

Yksikkötestaus hyödyntää lähes yksinomaan rakenteellisia testausmenetelmiä ja keskittyy yksikön sisältämien loogisten suorituspolkujen testaamiseen. Testauksen suorittaa useimmiten ohjelman tekijä, jolla on ohjelmakoodiin perustuvaan testaukseen tarvittava osaaminen ja tieto yksikön rakenteesta. Ongelmana on, että virheiden tekijällä on usein pienin todennäköisyys niiden löytämiseen [Hum89].

Olio-ohjelmoinnissa usein yksikkötestauksen pienin suoritettava yksikkö on luokan olio, mutta todellisuudessa viestit lähetetään yksittäiselle metodille, jolloin voidaan puhua metoditason testauksesta. Testattava yksikkö voi koostua luokasta, useista toisiinsa liittyvistä luokista (klusteri) tai suoritettavasta binääritiedostosta. Tyypillinen testauksen kohde on toisiinsa liittyvien luokkien muodostama klusteri [Bin00]. Olio-ohjelmoinnin erityispiirteistä erityisesti periytyminen lisää yksikkötestauksen resurs-

sitarvetta, sillä aliluokan toimivuuden varmistaminen edellyttää yliluokan ominaisuuksien testaamista myös aliluokan yhteydessä [Bin00].

2.2.2 Integrointitestaus

Integrointitestauksessa yksikkötestauksen läpäisseet ohjelmistokomponentit kootaan systemaattisesti alijärjestelmiksi ja nämä taas kokonaisiksi järjestelmiksi. Integrointitestaus alkaa silloin, kun kaksi tai useampia yksiköitä on valmiina integroitaviksi, ja päättyy, kun järjestelmä on valmis järjestelmätestaukseen. Integrointitestauksessa varmistetaan komponenttirajapintojen yhteensopivuus sekä se, että tietojen ja kontrollin välitys rajapintojen yli toimii oikein [EiR96]. Löydetyt virheet poikkeavat yksikkötestausvaiheessa löydetyistä ja liittyvät testattavien yksiköiden väliseen yhteistoimintaan ja riippuvuuksiin. Integrointitestausvaiheessa hyödynnetään sekä toiminnallisia että rakenteellisia testausmenetelmiä.

Komponenttien integroimiseen on olemassa useita strategioita. Ylhäältä alas (top-down) –integroinnissa testaus aloitetaan kontrollihierarkian ylimmästä yksiköstä, jonka kutsumat yksiköt korvataan tyngillä ja yksikkö testataan. Onnistuneen testauksen jälkeen tyngät korvataan seuraavan tason yksiköillä. Näiden kutsumia yksiköitä simuloidaan tyngillä ja syntynyt kokonaisuus testataan. Integrointia jatketaan kasvattaen testattavaa yksikköä pala palalta, kunnes koko järjestelmä on koottu yhteen ja testattu. Tämän lähestymistavan käyttöä haittaa se, ettei aina ole olemassa yhtä yksittäistä ylhäältä alas johtavaa polkua, jonka pohjalta integroinnin voisi suorittaa [Bei84]. Lisäksi tynkien tuottamisen aiheuttamat kustannukset voivat nousta suuriksi. Tyngät saattavat sisältää hyvin monimutkaisia rutiineja, minkä johdosta ne voivat olla hankalia ja aikaa vieviä toteuttaa. Myös tyngät tulisi testata niiden oikean toiminnallisuuden varmistamiseksi. Beizer suosittaakin tynkien käyttöä vain niissä tapauksissa, joissa ne joko ovat hyvin yksinkertaisia rakentaa tai aikataulu pakottaa korvaamaan niillä aidot ohjelmistoelementit [Bei84].

Ylhäältä alas etenevän integroinnin etuna on mahdollisuus varmistaa kontrollirakenteiden toiminta jo testauksen alkuvaiheessa [Bei84, Bin00]. Lisäksi testaus ja integrointi voidaan aloittaa hyvin aikaisessa vaiheessa. Komponentteja voidaan kehittää rinnakkain ja tarvittaessa voidaan lykätä alemman tason laiteriippuvien komponent-

tien valmistamista. Tämä on erityisen hyödyllistä, mikäli laiteympäristö on vasta kehitysvaiheessa tai sen muuttuminen kehitystyön aikana on mahdollista [Bin00].

Alhaalta ylös (bottom-up) -integrointi aloittaa testauksen kutsuhierarkian alimmista yksiköistä tai komponenteista, jotka integroidaan yhteen klustereiksi. Testauksen jälkeen klustereihin liitetään uusia klusteriin kuuluvia yksiköitä kutsuvia yksiköitä, ja syntyneet uudet klusterit testataan. Klustereiden kokoamista ja testausta jatketaan, kunnes koko sovellus on koottu. Puhtaimmillaan tämä strategia edellyttää perusteellista yksikkötestausta kullakin integrointitasolla [Bei84]. Lisäksi tämän strategian haittana on erillisten testiajureiden tarve. Kukin integrointikierros vaatii useita ajureita, jolloin ajureiden aiheuttamat kustannukset voivat nousta merkittäviksi. Näitä kustannuksia voidaan pienentää, mikäli ajurit suunnitellaan uudelleenkäytettäviksi [Bin00]. Myös kontrolli- ja arkkitehtuuriongelmien paljastuminen vasta hyvin myöhäisessä vaiheessa voi olla ongelmallista [Bei84, Bin00].

Alhaalta ylös -integrointi mahdollistaa yksiköiden rinnakkaisen toteuttamisen ja testauksen [Bei84]. Lisäksi järjestelmän toiminnan kannalta kriittistä toiminnallisuutta (muistirajoitteet, suorituskyky tai matalan tason laiterajapinta, johon liittyy kompleksista tilakäyttäytymistä) toteuttavien komponenttien toimivuus voidaan varmistaa jo aikaisessa vaiheessa [Bin00].

Kertarysäys (big bang) ei ole varsinainen määritelty integrointistrategia, mutta todellisuudessa varsin yleisessä käytössä yksiköiden integroinnissa [Bei84]. Kertarysäyksessä ohjelman kaikki yksiköt kootaan kerralla yhteen ja intergrointitestaus aloitetaan edellyttäen, että kokonaisuus saadaan toimimaan. Menetelmän etuina on tynkien ja ajurien tarpeettomuus sekä tietyissä hyvin rajatuissa tilanteissa mahdollisuus saada integrointitestausvaihe päätökseen nopeasti. Kertarysäys soveltuu käytettäväksi silloin, kun testattavana on pieni ja hyvin strukturoitu järjestelmä, jonka komponentit on testattu huolellisesti, kun olemassa olevaan järjestelmään on tehty muutamia muutoksia tai kun järjestelmä rakennetaan kokoamalla yhteen aiemmin käytössä olleita, luotettaviksi todettuja komponentteja. Yleensä kertarysäystä ei kuitenkaan suositella, sillä siinä virheiden löytäminen on erittäin työlästä ja aikaa vievää. Havaitut virheet voivat käytännössä sijaita millä tahansa integroitujen komponenttien rajapin-

nalla ja lisäksi monet virheet voivat jäädä kokonaan havaitsematta ja siirtyä siksi järjestelmätestiin [Bin00].

Beizer suosittaa sellaista integrointia, jossa ohjelman ydinohjelma (backbone), eli keskeiset toiminnalliset elementit, joilla ohjelma minimissään saadaan toimimaan, koostetaan ensin yhteen kertarysäyksellä. Tämä toiminnallinen ydin testataan huolella, jolloin sitä voidaan jatkossa käyttää ensisijaisena testialustana. Tämän ydinohjelman toiminnallisuutta lisätään vähitellen integroimalla siihen yhä uusia yksiköitä ja testaamalla koko ajan kasvavaa kokonaisuutta [Bei84]. Beizerin menetelmän käyttö edellyttää kuitenkin huolellista järjestelmän rakenteen ja riippuvuuksien analysointia ydintoiminnallisuuden selvittämiseksi. Binder suosittaa ydinohjelmaintegroinnin käyttöä erityisesti sulautettujen järjestelmien integrointitestauksessa [Bin00].

2.2.3 Järjestelmätestaus

Järjestelmätestausvaiheessa varmistutaan järjestelmän (ohjelma, laitteisto, tietokannat, tietoliikenne ja ulkoiset liittymät) toimivuudesta kokonaisuutena, sillä yksittäisten komponenttien erillinen verifiointi ei vielä takaa oikein toimivaa järjestelmää. Järjestelmätason virheet voivat johtua joko puuttuvasta toiminnallisuudesta tai sellaisesta vuorovaikutuksesta, jota ei ole mahdollista saada aikaan ennen kuin komponentteja testataan aidossa ympäristössä [Bin00].

Järjestelmätestausvaiheessa suoritetaan lisäksi ohjelman stressi-, kuormitus-, suorituskyky-, toipumis- sekä tietoturvallisuustestaus [Bei84]. Tyypillisiä järjestelmätestivaiheessa paljastuvia ongelmia ovatkin heikko yleissuorituskyky, alhainen läpimenokapasiteetti, riittämätön toipuminen ulkoisista poikkeustilanteista tai tietoturvan ongelmat. Järjestelmätestaus suositellaan suoritettavaksi jonkun ohjelman tuottamisesta riippumattoman tahon toimesta, jotta tietämys ohjelman rakenteesta tai yksityiskohdista ei vääristä testausta [Bei84] ja näin heikennä sen laatua.

2.2.4 Hyväksymistestaus

Hyväksymistestausvaiheessa asiakas ja järjestelmän käyttäjät varmistuvat sovelluksen käyttökelpoisuudesta. Useimmiten hyväksymistestaus suoritetaan asiakkaan toi-

mesta oikeassa tuotantoympäristössä asiakkaan tiloissa (niin sanottu beta-testaus). Toisena vaihtoehtona (niin sanottu alfa-testaus) hyväksymistestaus voidaan suorittaa asiakkaan toimesta toimittajan tiloissa. Hyväksymis- ja järjestelmätestaus voidaan myös tarvittaessa yhdistää. Hyväksymistestausvaiheessa ilmenevät ongelmat liittyvät tyypillisesti sovelluksen käytettävyyteen [Pre00].

3 Virheiden luokittelu

3.1 Vika – virhe – häiriö

Virheistä keskusteltaessa on tärkeää sopia yhtenäisestä termistöstä. Tässä tutkielmassa käytän vian, virheen ja häiriön määrittelyn pohjana IEEE:n terminologiastandardia (IEEE Standard Glossary of Software Engineering Terminology, v. 1990) ja British Computer Society:n testaussanastoa (Glossary of Testing Terms, v. 1998).

Vika (error) on ihmisen toimintaa, joka tuottaa virheellisen lopputuloksen [IEE610.12, BS7925-1]. Kyseessä voi olla vika kommunikaatiossa, virheellinen ajatusprosessi, typografinen erehdys, erheellinen käsitys ohjelmointikielen syntaksista tai muu inhimillinen syy, jonka seurauksena suunnitelmaan, dokumenttiin tai ohjelmakoodiin syntyy **virhe (fault)**.

Virhe (fault) puolestaan on vian (error) manifestaatio eli ilmentymä ohjelmassa [BS7925-1]. Ohjelmakoodissa se voi tarkoittaa virheellistä tai puuttuvaa ohjelmakoodin osa, toimenpidettä, prosessia tai tiedon määrittelyä [IEE610.12]. Mikäli virheen sisältävää ohjelman osaa suoritetaan, virhe saattaa aiheuttaa **häiriön (failure)** ohjelman suorituksessa. Ohjelmakoodin ohella myös ohjelman arkkitehtuurisuunnitelma, tekniset suunnitelmat tai dokumentaatio voivat sisältää virheitä. Suunnittelutason virheet voivat ohjelmankehitysprosessin edetessä muuttua ohjelmakoodin virheiksi.

Häiriö (failure) tapahtuu silloin, kun järjestelmä tai komponentti ei pysty suorittamaan vaadittuja toimenpiteitä tiettyjen suorituskykyvaatimusten puitteissa [IEE610.12]. Käyttäjän havaitsema toiminta siis poikkeaa siitä, mitä käyttäjä palve-

lulta odottaa [BS7925-1]. Häiriöitä ovat esimerkiksi virheet ohjelman lopputuloksessa, puutteet suorituskyvyssä, ohjelman kaatuminen suorituksen aikana tai se, että ohjelma ei lopeta aloittamaansa prosessointi.

Vian, virheen ja häiriön välinen mahdollinen syy-seuraussuhde on seuraava:

Vika → Virhe → Häiriö

Syy-seuraussuhdetta ei kuitenkaan aina ole, eikä vian, virheen ja häiriön välinen suhde ole yhden suhde yhteen [ChY96], vaan huomattavasti monimutkaisempi. Yksi vika, esimerkiksi tekijän erheellinen käsitys ohjelmointikielen syntaksista, voi aiheuttaa ohjelmakoodiin useita virheitä. Toisaalta useat ohjelman käyttäjän näkökulmasta erilaiset häiriötilanteet voivat johtua samasta virheestä. Vastaavasti yhden häiriön syynä voi olla useita eri virheitä. Virhe ei myöskään välttämättä koskaan ilmene häiriönä ohjelman toiminnassa. Esimerkiksi tilanteessa, jossa virhe sijaitsee kommenttirivillä tai sellaisessa osassa ohjelmakoodia, jota ei koskaan suoriteta, virhe ei ilmene häiriönä [Hum89]. Virhe-käsitteeseen liittyviä tulkinnanvaraisuuksia käsittelen enemmän luvussa 5.2.2.

3.2 Perusteet virheiden luokittelulle

Virheiden luokittelu luo pohjan testausprosessin eri sidosryhmien väliselle kommunikoinnille. Luokittelu ja sen mahdollistama virheiden analysointi antavat tietoa ohjelmatuotteen senhetkisestä tilasta sekä mahdollisista projektin ongelmakohdista. Lisäksi tietojen vertailu useiden eri projektien tai jopa eri organisaatioiden välillä mahdollistaa organisaation tai jopa koko teollisuudenalan ongelma-alueiden tunnistamisen [IEEE1044.1].

Virheiden luokittelu virhetyypin mukaan on tarpeen myös yrityksen ohjelmistotuotantoprosessin kehittämiseksi. Virheiden jako tyyppeihin ja näiden tyyppien esiintymistiheyden seuraaminen mahdollistaa resurssien keskittämisen sekä korjaavat toimenpiteet. Koulutuksella, uusien kontrollien luomisella, dokumentaation kehittämisellä tai katselmointien hyväksikäytöllä voidaan ennaltaehkäistä usein esiintyviä virhetyyppejä. Virheiden kerääminen ja jaottelu jonkin rationaalisen luokittelun mukaan

kertoo myös nykyisten testaustekniikoiden tehokkuudesta [Bei90], ja auttaa näiden testaustekniikoiden muokkaamisessa paremmin tarpeita vastaaviksi.

Virheiden luokitteluun ei ole olemassa mitään yleispätevää sääntöä, vaan organisaation tulee omista lähtökohdistaan harkita, millainen luokittelu parhaiten tukee yrityksen ohjelmistotuotantoprosessia ja sen kehittämistä yrityksen omat lähtökohdat huomioiden [Pre00]. Jonkinlainen luokittelu on kuitenkin tarpeen ottaa käyttöön. Käytettyjen luokkien tulee olla selvärajaisia ja selkeästi toisistaan erottuvia, jotta luokittelu olisi suhteellisen yksinkertaista ja näin vähemmän altista inhimillisille tulkinnoille. Lisäksi virheluokkia tulisi olla suhteellisen pieni määrä, jotta virheen luokittelu ja raportointi olisi helppoa [Bei84].

Esittelen kaksi virheiden luokittelujärjestelmää: IEEE:n standardin 1044 ohjelmapoikkeamien (software anomalies) luokitteluun ja IBM:ssä kehitetyn ortogonaalisen virheluokittelun (orthogonal defect classification, ODC). Kumpaankin luokittelujärjestelmään sisältyy joukko erilaisia ominaisuuksia, joilla ohjelmassa esiintyviä virheitä voidaan luonnehtia.

3.3 IEEE-standardi 1044 ohjelmistoanomalioiden luokittelusta

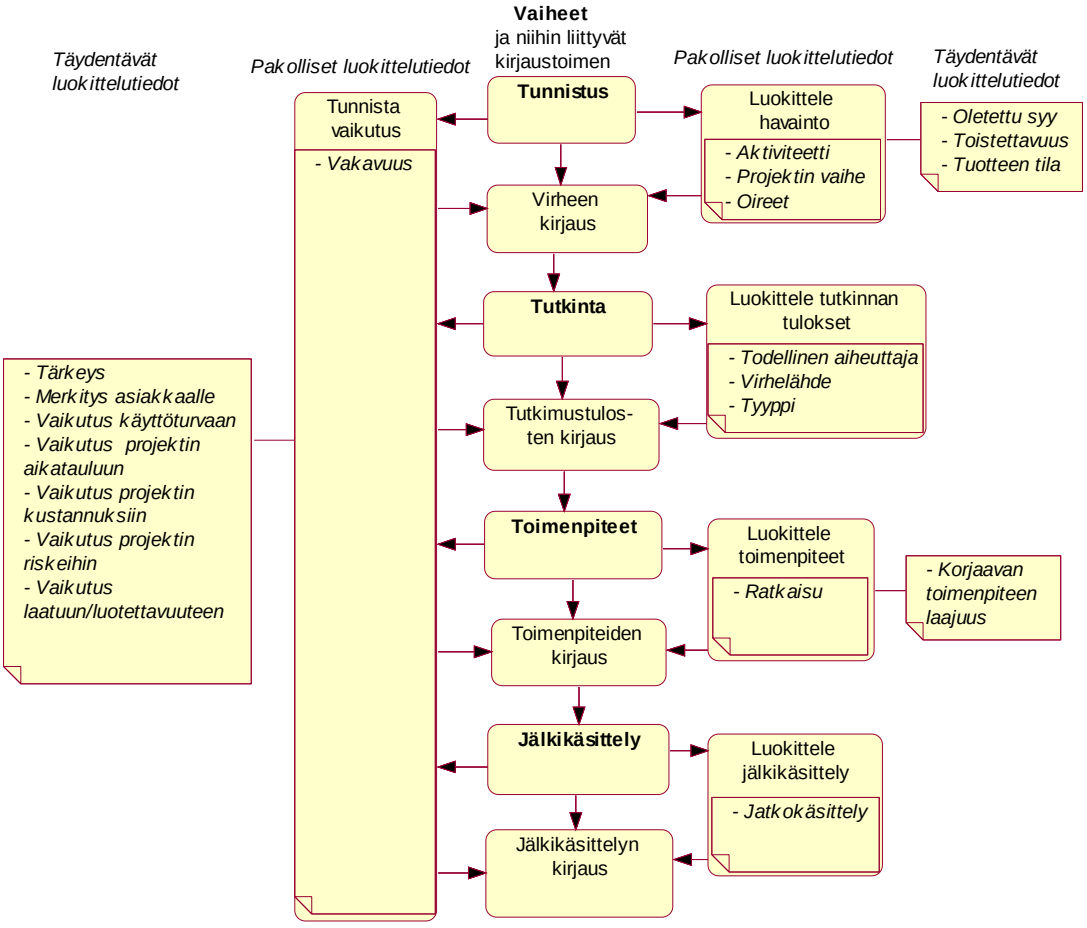
IEEE:n standardi 1044 vuodelta 1993 Ohjelmistoanomalioiden luokitteluun (Standard Classification for Software Anomalies) tarjoaa kattavan luokittelujärjestelmän ohjelmissa esiintyville poikkeamille (anomaly). Luokittelu ei koske pelkästään ohjelmistovirheitä, vaan kaikkia sellaisia ohjelman ominaisuuksia, jotka poikkeavat odotetusta. Tällaisia ovat ohjelmistovirheiden lisäksi esimerkiksi ohjelman kehitystoiveet. Standardi sopii käytettäväksi niin turvallisuuskriittisiin kuin kaupallisiinkin ohjelmatuotteisiin [IEEE1044].

3.3.1 Luokittelun vaiheet

IEEE:n anomalioiden luokittelustandardi kiinnittyy luokitteluprosessiin, joka kattaa poikkeaman koko elinkaaren löytöhetkestä tutkinnan ja toimenpiteiden kautta jälkikäsittelyyn. Poikkeaman ominaisuuksia täydennetään koko prosessin ajan. Standardi määrittelee joukon pakollisia luokittelutietoja. Ne muodostavat minimivaatimuksen

vertailukelpoiselle luokittelulle projektien tai organisaatioiden välillä. Pakollisten ominaisuuksien lisäksi luokittelu sisältää suuren joukon täydentäviä luokittelutietoja, joita yritys voi valintansa mukaan sisällyttää organisaation käyttöön valittavaan luokittelujärjestelmään [IEEE1044].

Varsinainen luokitteluprosessi muodostuu neljästä vaiheesta: tunnistus, tutkinta, toiminta, jälkikäsittely. Kunkin vaiheen sisällä toteutetaan kolme toimenpidettä: kirjaus, luokittelu ja vaikutuksen arviointi. Standardin tarjoama luokitteluprosessi on malli, jonka soveltuvuutta yrityksen toimintaan tulee tarkastella yrityksen omista tarpeista ja lähtökohdista alkaen [IEEE1044]. Kuva 2 havainnollistaa luokitteluprosessin eri vaiheisiin kytkeytyviä luokittelutietoja.



Kuva 2 **IEEE1044-standardin mukainen ohjelmistopoikkeaman luokitteluprosessi sekä eri vaiheiden pakolliset ja täydentävät luokittelutiedot**

Tunnistusvaiheen alussa ohjelmapoikkeama löydetään. Löytäjä voi olla kuka tahansa ohjelman kehityksen, käytön, ylläpidon tai arvioinnin kanssa tekemisissä oleva hen-

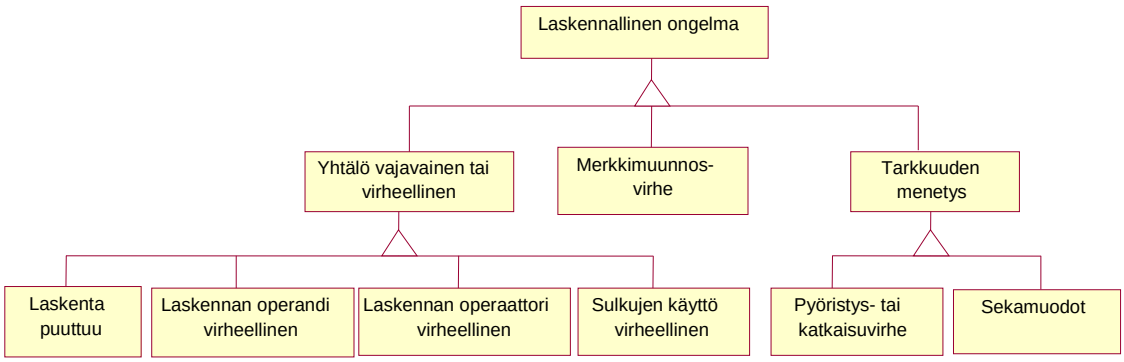
kilö huolimatta siitä, missä ohjelmakehityksen vaiheessa tuote on. Tunnistusvaiheessa kirjataan aluksi ylös anomalian tunnistamista edistävät tiedot, kuten esimerkiksi kuvaus, toimintaympäristö, alkuperä, laukaiseva tekijä ja ilmenemisajankohta. Havainto luokitellaan. Pakollisia tietoja ovat poikkeaman laukaissut työvaihe, projektin vaihe sekä anomalian aiheuttamat oireet. Poikkeama voidaan tarvittaessa luokitella myös oletetun syyn, toistettavuuden ja tuotteen tilan mukaan [IEEE1044].

Viimeisenä tunnistusvaiheessa poikkeaman löytäjä määrittelee oman käsityksensä perusteella virheen vaikutukset. Vähintään löytäjän on luokiteltava virheen vakavuus (severity). Vakavuus on objektiivinen, tekninen näkemys virheen vaikutuksista. Lisäksi löytäjä voi tallentaa tietokantaan myös oman, subjektiivisen käsityksensä poikkeaman tärkeydestä (priority), joka puolestaan mittaa sitä, kuinka tärkeää poikkeaman korjaus on projektin onnistumiselle. Muita mahdollisia luokiteltavia ominaisuuksia ovat poikkeaman merkitys asiakkaalle, sen vaikutus käyttöturvallisuuteen tai projektin aikatauluun, kustannuksiin, riskiin tai laatuun. Virheen vaikutustietoja päivitetään prosessin jokaisessa vaiheessa [IEEE1044].

Tutkimusvaiheessa anomaliaan perehdytään tarkemmin. Sen olemassaolo varmistetaan ja todennetaan, että se voidaan toistaa. Anomalian poistomahdollisuudet selvitetään. Samalla analysoidaan, mihin kaikkialle mahdollisesti tarvittavat muutokset vaikuttavat. Tutkimuksen perusteella voidaan myös päätyä toteamaan, ettei asia vaadi toimenpiteitä. Tutkimusvaiheessa anomalia luokitellaan todellisen aiheuttajan (tuote, testijärjestelmä, alusta, kolmannen osapuolen toimittama osa, käyttäjä tai tuntematon), virhelähteen (määrittely, ohjelmakoodi, tietokanta, manuaalit ja oppaat, suunnitelmat ja proseduurit, raportit tai standardit) sekä tyypin mukaan. Tutkimusvaihe päättyy tekijän tekemään anomalian vaikutusten luokittelun tarkistamiseen ja mahdolliseen tietojen päivitykseen [IEEE1044].

Poikkeaman tyyppi kertoo, onko anomalia ohjelman dokumentaatiossa, laajennuksessa tai jossakin ohjelmakoodin osassa. Tällaisia ohjelmakoodin osia ovat esimerkiksi suorituslogiikka, rajapinnat, tiedonkäsittely ja laskenta. Tyyppiluokittelussa on useita hierarkkisia tasoja, joten käytettävän luokittelun tarkkuus voidaan valita kullekin organisaatiolle ja projektille parhaiten sopivaksi. Ylimmällä tasolla anomaliat luokitellaan johonkin kahdeksasta perustyyppistä: looginen, laskennallinen, rajapin-

ta/ajoitus, tiedonkäsittely-, data-, dokumentointi-, dokumentin laatuongelma tai kehitysehdotus [IEEE1044]. Kuva 3 havainnollistaa virhetyypin hierarkkista rakennetta.



Kuva 3 Esimerkki IEEE1044-standardin tyyppiluokitteluhierarkiasta

Tutkimusvaiheen perusteella päätetään toimenpiteistä. Anomalian käsittelemiseksi tarvitaan välittömiä toimenpiteitä. Niiden lisäksi tarvitaan toimenpiteitä, joilla muokataan prosesseja, toimintapolitiikkaa tai muita olosuhteita siten, ettei vastaavia poikkeamia enää esiintyisi. Välittömistä toimenpiteistä voidaan kirjata muistiin tiedot ratkaisun identifioinnista, sen toteutuksesta sekä muista varsinaisen perussyyn korjaamiseen tähtäävistä toimista. Pakollisena luokittelutietona tallennetaan tieto ratkaisusta (korjaus heti, korjaus seuraavaan versioon, lykätty, ei korjata). Mikäli on olemassa toimenpiteitä, joilla anomalian esiintyminen vastaisuudessa voidaan estää, myös nämä voidaan luokitella. Kyseessä voi tällöin olla osastotason toimenpide, esimerkiksi ohjelmointikoulutus tai resurssien uudelleenryhmittely. Koko yrityksen tasoinen toimenpide voi olla esimerkiksi prosessien tarkistaminen. Tutkimus- tai koulutusohjelman sponsorointi on koko teollisuudenalaa koskeva toimenpide. Kyseessä voi olla myös koko tutkimus- ja koulutusalaa koskeva ongelma, jonka ratkaiseminen vaatii esimerkiksi uuden teknologian kehittämistä. Tutkimusvaiheen viimeisenä toimenpiteenä on anomalian vaikutusten tarkistus ja mahdollinen päivitys [IEEE1044].

Jälkikäsittelyvaiheessa, kun kaikki tarvittavat korjaustoimet on suoritettu ja/tai tarvittavat pitkäaikaiset toimenpiteet tunnistettu, poikkeamasta voidaan kirjata muistiin jälkikäsittelytoimenpiteet ja niiden verifiointi. Anomalia luokitellaan valitun jatkokä-

sittelytavan mukaan joko suljetuksi, lykätyksi, sulautetuksi toiseen ongelmaan tai siirretyksi toiseen projektiin. Anomalian vaikutustiedot tarkistetaan ja päivitetään tarvittaessa [IEEE1044].

3.3.2 IEEE1044-luokittelun käyttöönotto

IEEE-standardin 1044 soveltamisohje tarjoaa selkeän käyttöönotto-ohjeen standardin mukaiselle luokittelulle. Mikäli yrityksellä on jo käytössään jonkinlainen luokittelujärjestelmä, sen sisältämät kategoriat tulee linkittää IEEE:n luokitteluun määrittelemällä käytössä olevan luokittelun vastaavuudet IEEE:n luokitteluun. Yrityksen tulee myös päättää, vaaditaanko standardia sovellettavaksi sellaisenaan, jolloin yrityksen tulee omaksua vähintään standardin pakollisiksi määrittelemät ominaisuusluokat sekä tarvitsemansa osat täydentävistä tiedoista. Mikäli standardia ei haluta soveltaa kokonaisuudessaan, yritys valitsee kaikista luokitteluryhmistä parhaiten omaan tilanteeseensa sopivat ja hyödyllisimmiksi katsotut ominaisuusluokat [IEEE1044.1].

Kaikista yrityksen luokitteluun valituista ominaisuuksista määritellään pakollisiksi ne, jotka ovat oleellisia yrityksen liiketoiminnalle juuri tällä hetkellä. Standardin soveltamisohje suosittaa virheiden luokittelun alussa minimoimaan pakollisten luokkien määrän. Kun organisaatio tottuu anomalioiden luokittelujärjestelmään ja ymmärtää kokonaisnäkemyksen tarpeellisuuden kehityksen mahdollistajana, pakollisten luokkien määrää voidaan lisätä [IEEE1044.1]. Saatujen tietojen pohjalta tehtyjen analyysien esittäminen antaa luokittelujen tekijöille selkeää palautetta tehdystä työstä ja motivoi heitä sekä itse luokitteluun että myös tarkkuuteen omaisuuksien luokittelussa.

Jokaisesta yrityksen luokittelujärjestelmään valitusta omaisuudesta päätetään lisäksi taso, jolla luokittelu suoritetaan. Luokittelu kannattaa aloittaa hierarkian ylimmästä tasosta. Poikkeamien testijoukko luokitellaan tällä karkealla yleistasolla, ja analysoidaan, onko luokittelu tarpeisiin nähden riittävä. Mikäli näin ei ole, siirrytään luokittelukategorian hierarkiassa alemmas, kunnes saavutetaan yrityksen tarpeita vastaava tarkkuustaso. Lisäksi organisaation ja projektien erityistarpeisiin tulee tarvittaessa kehittää luokkien sisälle uusia ryhmiä (classifications) [IEEE1044.1].

Käyttöön valitut ominaisuuskategoriat ja näiden sisältämät ryhmät dokumentoidaan, eli yrityksen käytössä olevalla terminologialla kuvataan, mitä kullakin ominaisuusluokalla ja luokan sisältämättä ryhmällä tarkoitetaan. Mikäli IEEE1044-standardin noudattaminen on yritykselle tärkeää, käytettävän luokittelun suhde standardiin dokumentoidaan esimerkiksi jäljitettävyysmatriisilla [IEEE1044.1].

Lisätiedot täydentävät poikkeavuuksien seurantajärjestelmää. Poikkeavuuteen liittyvä tekstitieto, kuten vaikkapa kuvaukset, muistiinpanot ja avainsanat, voi olla erityisen hyödyllistä poikkeamaa analysoitaessa. Poikkeavuuteen voidaan liittää erilaisia tunnistusta helpottavia tietoja, kuten ohjelman versionumero, anomalian tunniste tai suorittimen tiedot. Anomaliaan liittyy myös erilaista mitattavaa tietoa, joka voi olla hyödyllistä tallettaa, kuten tapahtumapäivämäärä, raportointipäivämäärä, korjauksen tuntimäärä, testauksen tuntimäärä tai korjattujen koodirivien määrä. Hyödyllistä voi olla myös liittää anomalian tietoihin muiden asiaan liittyvien anomalioiden tunnisteet. Anomaliaan liittyvä dokumentaatio sekä oireloki, testiloki, testitapaus tai kuvaruutunäytöt voidaan myös tallettaa luokitteluun. Lisäksi erilainen hallinnollinen informaatio, kuten esimerkiksi tiedot anomalian käsittelyvaiheesta, ilmoittajasta, korjaajasta tai testaajasta voidaan liittää poikkeaman kirjauksen yhteyteen. Minkään edellä mainittujen lisätietojen käyttöä standardi 1044 ei edellytä, mutta yrityksen kannattaa virheluokittelun kokonaismäärittelyn yhteydessä päättää, mitä näistä lisätiedoista halutaan systemaattisesti kerätä. Myös nämä tiedot tulee dokumentoida yrityksen omalla terminologialla [IEEE1044.1].

Luokiteltavien ominaisuuksien valinnan jälkeen selvitetään, missä vaiheessa poikkeaman käsittelyprosessia kunkin luokittelukategorian tieto on saatavissa ja luokiteltavissa. Tämän jälkeen luokittelu kytketään osaksi yrityksen soveltamaa anomalian käsittelyprosessia. Viimeisenä käyttöönoton vaiheena luokittelun käyttäjät ja johto koulutetaan. Koulutuksella käyttäjät paitsi opastetaan luokittelun käyttöön, myös motivoidaan säännöllisyyteen, tarkkuuteen ja täydellisyyteen luokittelussa. Johtoa puolestaan informoidaan siitä, miten kerättyjä tietoja on tarkoitus käyttää ja analysoida, jolloin varmistetaan luokittelulle myös johdon tuki [IEEE1044.1].

3.3.3 Luokitusten hyödyntäminen

Poikkeamien luokittelutietojen tallennus helppokäyttöiseen relaatiotietokantaan mahdollistaa raporttien, tilastojen ja erilaisten kehitysanalyyseissa tarvittavan tiedon helpon tuottamisen. Tietovarastoon kertyvän informaation perusteella voidaan selvittää muun muassa erilaisten virhetyyppien yleisyyttä, eri aktiviteettien tehoa virheiden löytäjänä tai kehitysvaiheita, joissa syntyy eniten virheitä. Luokittelun suunnittelun yhteydessä tuleekin miettiä kerättävän tiedon käyttöä, tallennusta sekä sitä, miten aineistosta analysoitu tieto välitetään niille, jotka tietoa tarvitsevat. Analysoitua tietoa voidaan käyttää projektin hallinnointiin, prosessin parantamiseen ja tuotteen arviointiin [IEEE1044.1].

Projekteissa saatuja luokittelutietoja voidaan hyödyntää muun muassa sen määrittelemiseksi, mitkä poikkeamat käsitellään seuraavaksi jakeluun menevässä versiossa ja mitkä voidaan siirtää myöhempiin ohjelmaversioihin. Jos poikkeamasta on luokiteltu sen vaikutus projektin kuluihin, riskeihin tai laatuun/luotettavuuteen, on helpompi päättää, korjataanko poikkeama ohjelmakehityksen viime hetkillä vai lykätäänkö korjausta myöhempään ajankohtaan [IEEE1044.1].

Prosessin parantamisessa huomioitavia luokittelukategorioita ovat tyyppi, virhelähde, todellinen syy, projektin vaihe ja aktiviteetti. Virhetyyppien syy-seuraussuhteen analysoinnilla voidaan yrityksen prosesseja tarvittaessa muuttaa tai keskittää resursseja niihin ohjelmakehityksen vaiheisiin, jotka analysoinnin perusteella vaikuttavat ongelmallisilta [IEEE1044.1].

Tuotteen osalta anomalioiden luokittelu auttaa päättämään, onko ohjelma valmis julkaistavaksi. Ohjelmasta löytyneiden anomalioiden määrä suhteutettuna niiden vakavuuteen sekä merkitykseen asiakkaalle toimivat hyvänä mittarina tuotteen kypsyydestä [IEEE1044.1].

3.3.4 Valmiin luokittelun tarjoamat edut

Valmiin standardin omaksumisen etuna ovat kustannussäästöt, sillä yrityksen ei tarvitse panostaa resursseja virheluokittelun pohtimiseen ja luomiseen [IEEE1044.1].

Laajalle levinneen standardin käyttö mahdollistaa myös vertailut muihin yrityksiin tai toimialan keskiarvoihin.

IEEE:n standardi 1044 soveltaminen tarjoaa hyvän viitekehyksen ohjelmistopoikkeamien käsittelylle yrityksessä. Standardissa poikkeaman elinkaari on selkeästi vaiheistettu, ja ominaisuuksien liittäminen poikkeamaan sen elinkaaren eri vaiheissa tarjoaa yksinkertaisen mallin toimivaksi käytännöksi myös aidoissa ohjelmistotuotantoympäristöissä. Eri vaiheet ja niihin liittyvät luokittelut on helppo muokata osaksi yrityksen omaa ohjelmistotuotantoprosessia, jolloin vastuu poikkeaman käsittelystä ja sen luokittelusta on selkeä. Näin päästään kattavaan luokittelujärjestelmään, jossa kaikki löydetyt poikkeamat käsitellään ja luokitellaan samalla tavoin. Tämä edesauttaa vertailukelpoisen tiedon saamista sekä päätöksentekoa että prosessin kehittämistä varten. Myös luokittelun käyttöönotto on selkeästi määritelty ja ohjeistettu, mikä houkuttaa sen omaksumiseen.

3.4 Ortogonaalinen virheluokittelu

Ortogonaalinen virheluokittelu (Orthogonal defect classification, ODC) on IBM:n tutkimuslaboratoriossa kehitelty, kirjallisuudessa Chillaregen ja kumppaneiden vuonna 1992 [CBC92] esittelemä käsite, jossa virheiden luokittelun avulla pyritään selittämään erilaisia syy-seuraussuhteita sovelluskehitysprosessissa ja tuotteen laadussa. Aiemmassa empiirisessä tutkimuksessaan Chillaregen, Kaolin ja Conditin [CKC91] kykenivät osoittamaan, että yksinkertaisella virheiden luokittelulla oli mahdollista saada tietoa kehitysprosessin ongelmista jo ohjelman kehitysvaiheessa. Välitön palaute mahdollistaa reagoinnin ongelmatilanteisiin jo yksittäisen ohjelmistotuotantoprojektin kuluessa, mikä on selkeä etu erilaisiin projektin jälkeen palautetta antaviin mittareihin. Edellytyksenä tälle on virheiden luokittelujärjestelmä, jolla erilaiset virheluokat suhteutetaan tuotteen kehitysprosessiin ja seurataan tuotteen edistymistä prosessin läpi.

ODC:n perusajatuksena on, että yksittäiseen virheeseen liitetään erilaisia ominaisuuksia. Ominaisuuksien kollektiivisen jakauman perusteella voidaan tehdä päätelmiä tuotteen kypsyydestä (maturity) prosessin eri vaiheissa, kypsyyden muutoksista

sekä prosessin eri vaiheiden toimivuudesta tuotteen kypsyyden lisääjänä. Varsinaisena mittarina toimii siis virheen ominaisuuksien jakaumassa tapahtunut muutos, joka kertoo kehityssuunnan. ODC pyrkii tarjoamaan prosessin sisäisen mittarin, jonka avulla virheistä saadaan avaintiedot ja voidaan mitata erilaisia syy-seuraussuhteita [Chi94].

ODC-luokittelun versiossa 5.1 erilaisia virheen ominaisuuksia on kahdeksan. Näistä kolme - aktiviteetti, laukaisin (Taulukko 4) ja asiakasvaikutus (Taulukko 1) - on tunnistettavissa heti virheen löydyttyä. Muut ominaisuudet - kohde, virhetyyppi, laatu, lähde ja historia - liitetään virhehavaintoon virheen korjauksen yhteydessä (Taulukko 2).

Asiakasvaikutus
Asennettavuus
Palvelevuus
Standardi
Eheys/tietoturva
Päivitettävyys
Luotettavuus
Suorituskyky
Dokumentaatio

Taulukko 1 Virheiden asiakasvaikutus-ominaisuuden luokittelu ODC:n versiossa 5.1 [BKS98]

Kohde	***Virhetyyppi***	***Laatu***	***Lähde***	***Historia***
Suunnittelu	Sijoitus/alustus	Puuttuva	Itse kehitetty	Pohja
Ohjelmakoodi	Tarkistus	Virheellinen	Uudellenkäytettävä	Uusi
	Algoritmi/metodi	Muu	kirjasto	Uudelleenkirjoitettu
	Funktio/luokka/olio		Ulkoistettu	Uudelleenkorjattu
	Ajoitus/Sarjallistaminen		Siirretty	
	Rajapinta/ OO-viestit			
	Yhteys			

Taulukko 2 Virheen korjauksen yhteydessä tunnistettavat ominaisuudet ODC-luokittelun versiossa 5.1 [BKS98]

Useiden ominaisuuksien avulla on mahdollista luokitella virheitä moniulotteisesti ja näin lisätä ymmärrystä sovelluskehitysprosessista. ODC-luokittelun käyttö ei kuitenkaan edellytä kaikkien ominaisuuksien käyttöönottoa, vaan näistä voidaan poimia tutkittavan prosessin kannalta olennaisimmat [ChP02]. Tässä tutkielmassa keskityn tarkastelemaan kolmea mielestäni keskeisintä luokituksen ominaisuutta: aktiviteettia, laukaisinta ja virhetyyppiä.

3.4.1 Aktiviteetti

Aktiviteetti viittaa siihen todelliseen ohjelmakehityksen vaiheeseen, jossa virhe havaittiin. Jakamalla prosessi vaiheisiin, aktiviteetteihin, luokittelusta saadaan sovellettavasta prosessimallista riippumaton. Geneerinen aktiviteettilista voisi sisältää esimerkiksi seuraavat aktiviteetit: arkkitehtuurin katselmointi, koodikatselmointi, yksikkötestaus, toiminnallinen testaus ja järjestelmätestaus [BKS98]. Toiminnallinen testaus on yleistermi, joka Chillaregen & Bassinin määritelmässä [ChB95] sisältää paitsi yksikkötestauksen, myös rajapintojen toimivuuden varmistamisen. Kunkin organisaation tulee kuitenkin muokata aktiviteettilista oman verifiointiprosessinsa ja sen osa-alueiden perusteella parhaiten vastaamaan omia tarpeitaan.

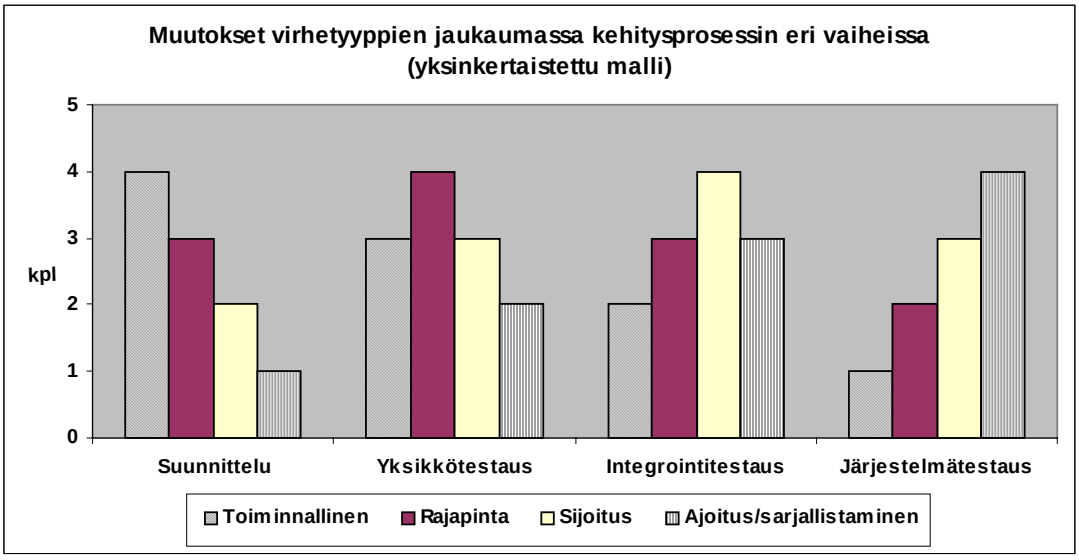
3.4.2 Virheen tyyppi

Virheen tyyppi kuvaa virheen luonnetta ja sen korjaustoimien vaikutusaluetta [BKS98]. Virheen tyypityksen suorittaa yleensä virheen korjaava ohjelmoija varsinaisen virheenkorjauksen yhteydessä. Tyypit ovat tarkoituksellisen yksinkertaisia ja niitä on pieni joukko, jotta valinta niiden välillä on helppoa. Tutkijat ovat virheiden luokittelun yhteydessä painottaneet virheen löytäjän mielentilan ja virheen löytötavan vaikutusta virheen luokitteluun [Bei90, Wey94]. Luokkien ortogonaalisuudella vähennetään juuri tätä luokitteluissa aina ilmenevää inhimillistä subjektiivisuutta ja virhemahdollisuutta tarjoamalla käyttöön virheluokat, jotka ovat selkeästi toisistaan erottuvia ja samalla toisensa poissulkevia, mutta jotka kattavat kuitenkin kaikki luokittelussa tarvittavat vaihtoehdot [CBC92].

Testattuaan luokitusta käytännössä tutkijaryhmä päätyi esittämään kahdeksaa erillistä virhetyyppiä: funktio- (function error), sijoitus- (assignment error), rajapinta- (interface error), tarkistusvirhe (checking error), ajoitus/sarjallistamisvirheet (timing/serialization error), käännös/pakkaus/liitosvirhe (build/package/merge error), dokumentointivirhe (documentation error) ja algoritmivirhe (algorithm error) [CKC91]. Virhetyypeistä on pyritty tekemään mahdollisimman yleiskäyttöisiä, jotta ne soveltuisivat käytettäväksi kaiken tyyppisissä ohjelmakehitysprosesseissa tuotteesta riippumatta [CBC92]. Sittemmin virhetyypeistä dokumentointi on siirretty

Asiakasvaikutukset-ominaisuuden joukkoon ja olio-ohjelmien myötä virheluokkia on täsmennetty (Taulukko 2).

Kussakin ohjelmakehityksen vaiheessa löydetyt virheet tyypitetään ja kunkin tyypin suhteellinen osuus kaikista kyseisessä vaiheessa löydetyistä virheistä lasketaan. Näin muodostuu kyseisen vaiheen virhetyyppien jakauma eli virheprofiili (defect type signature). Virheprofiilin tulkitaan kuvastavan tuotteen kasvavaa vakautta ja luotettavuutta sovelluskehitysprosessin edetessä [BKS98]. Löydettyjen virheiden jakauman tulisi kussakin kehitysprosessin vaiheessa olla kyseiselle vaiheelle tyypillinen ja noudattaa tiettyä trendiä prosessin edistyessä (Kuva 4). Yksittäisessä projektissa ilmenevät poikkeamat vaiheen tavanomaisesta virhejakaumasta osoittavat mahdollisia ongelmakohtia projektin kulussa.



Kuva 4 **Yksinkertaistettu malli virhetyyppien jakaumassa tapahtuvasta muutoksesta sovelluskehitysprosessin eri vaiheissa [CBC92]**

Tutkijoiden mukaan kukin virhetyyppi syntyy tyypillisimmin tietyssä ohjelmakehitysprosessin vaiheessa. Eri virhetyyppien tulisi myös löytyä prosessin eri vaiheissa. Mikäli esimerkiksi järjestelmätestausvaiheessa löytyy paljon toiminnallisia virheitä, joiden tulisi normaalissa kehitysprosessissa suurimmalta osaltaan löytyä jo arkkitehtuurin katselmoinnissa, viittaa havainto siihen, että arkkitehtuurin katselmoinnissa ei jostakin syystä kyseisen projektin osalta ole päästy tavoitteisiin.

Taulukko 3 esittää ODC-luokittelun mukaisen virhetyyppien ja sovelluskehitysprosessin välisen suhteen. Ympyrä osoittaa kyseisen virhetyypin tavallisimman syntypaikan kehitysprosessissa, rastit puolestaan ne prosessin vaiheet, joissa kyseisen tyyppisten virheiden tulisi tutkijoiden mukaan normaalioloissa löytyä.

Perusassosiaatiotaulukko

	Virhetyyppi			
Vaihe	Toiminnallinen	Tarkistus	Ajoitus	Algoritmi
Arkkitehtuurisuunnittelu	**o**			
Alemman tason suunnittelu			**o**	
Koodaus		**o**		**o**
Arkkitehtuurin katselmointi	**x**			
Alemman tason suunnitelman katselmointi			**x**	
Koodikatselmointi		**x**		**x**
Yksikkötestaus		**x**		**x**
Toiminnallisuuden testaus	**x**			**x**
Järjestelmätestaus			**x**	

Taulukko 3 Eri virhetyyppien tavanomaisimmat synty- ja löytöpaikat [CBC92]

On kuitenkin huomattava, että ennen varsinaisten johtopäätösten tekoa eri virhetyyppien jakauma on ehdottomasti kalibroitava käytettävään sovelluskehitysprosessiin. Tutkijoiden mukaan on kuitenkin mahdollista hyödyntää luokittelua jo ennen kalibrointitiedon valmistumista prosessin oikean kehityssuunnan osoittajana [CBC92].

3.4.3 Virheen laukaisin

Tärkeä virheen ominaisuus ODC-luokittelussa on virheen laukaisin (trigger). Laukaisin on tekijä tai olosuhde, jonka seurauksena virheellinen ohjelman osa suoritetaan ja virhe ilmenee [SuC91]. Myös laukaisinten jakauma muuttuu ohjelmistotuotantoprosessin aikana ja tarjoaa tutkijoiden mukaan menetelmän arvioida tuotteen verifiointiprosessia. Chillarege ja Bassin kuitenkin painottavat laukaisinten arvojoukon kokeellista verifiointia kyseisen ohjelmistotuotantoprosessin tarpeisiin. Heidän mukaansa tarvitaan useiden vuosien systemaattista tietojen keruuta, kokeilua sekä useita kokeiluprojekteja, jotta prosessiin sopiva ja sitä parhaiten palveleva arvojoukko saadaan muodostettua [ChB95]. Toisaalta kun arvojoukko on selvillä ja kalibroitu kyseiseen prosessiin, erilaisten jakauma- ja tulostietojen tuottaminen on varsin yksinkertaista.

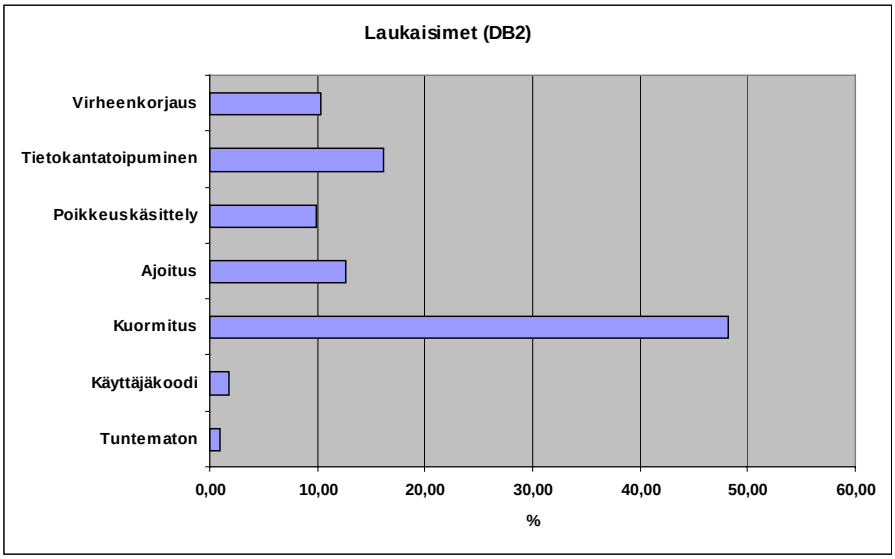
Toisin kuin virhetyypit, joiden luokat säilyvät samoina koko tuotteen elinkaaren ajan, laukaisimet ovat erilaisia ohjelmistotuotantoprosessin eri vaiheissa (Taulukko 4). Yhteistä kaikille laukaisimille kuitenkin on, että ne yrittävät jäljitellä asiakkaan suorittamaa ohjelman käyttöä kyseiselle verifiointivaiheelle ominaisella tavalla [ChB95].

Aktiviteetti:

Katselmointi	**Yksikkötesti**	**Toiminnallinen testi**	**Järjestelmätesti**
Suunnittelun vastaavuus	Yksinkertainen polku	Testikattavuus	Kuormitus/stressi
Logiikka/tietovirta	Kompleksinen polku	Testien vaihtelevuus	Käynnistys/ uudelleen-käynnistys
Komponenttien yhteensopivuus		Testien järjestys	Toipuminen/ Poikkeus
Yhteensopivuus aiempiin		Testien vuorovaikutus	Laitteistokokoonpano
Kieliriippuvuus			Ohjelmistokokoonpano
Rinnakkaisuus			Estynyt testaus
Sivuvaikutukset			
Harvinaiset tilanteet			

Taulukko 4 Ohjelmakehityksen eri vaiheissa (aktiviteetti) ilmenevät virheen laukaisimet ODC-luokittelun versiossa 5.1 [BKS98]

Tietoa laukaisinten jakaumasta prosessin eri vaiheissa voidaan käyttää hyväksi testaus- ja muiden verifiointiresurssien oikeassa kohdentamisessa. Esimerkkinä Kuva 5 esittää erään tietokantatuotteen virheiden laukaisimet tuotantokäytöstä kolmen vuoden ajalta. Laukaisinten jakaumasta on helppo nähdä, että kyseisen tuotteen kohdalla kuormituksen testaus on erityisen keskeistä.



Kuva 5 DB2-tietokantatatuotteen laukaisinten jakauma tuotantokäytöstä kolmen vuoden ajalta [CBC92]

Chillarege ja Bassin havaitsivat virheiden laukaisinten jakauman muuttuvan myös ajan funktiona. He tutkivat kahden vuoden ajalta julkistamisen jälkeen eräästä käyttöjärjestelmästä raportoituja virheitä. Laukaisinten jakaumassa havaittuja muutoksia voidaan hyödyntää resurssien suuntaamisessa keskittymällä juuri ennen julkaisua sellaisten virheiden etsimiseen, joiden laukaisinten piikki osuu välittömästi julkaisun jälkeiseen ajankohtaan. Tällaisia laukaisimia ovat esimerkiksi dokumentaatiossa olevat virheet. Toisaalta sellaisten laukaisinten, joiden tiedetään tyypillisesti esiintyvän vasta pidemmän käyttöajan jälkeen, mittavampaa testausta voidaan tietoisesti lykätä esimerkiksi johonkin korjausversioon [ChB95].

3.4.4 ODC-luokittelun käyttöönotto

Huolellinen virhetyyppien ja –laukaisinten tekninen luokittelu on ensimmäinen ja tärkein vaihe ODC-luokittelumallin soveltamisessa ohjelmistoprosessiin. Tehtyä luokittelua tulee testata empiirisesti useissa kokeiluprojekteissa, ja sitä on kalibroitava saatujen tulosten perusteella olosuhteisiin sopivaksi.

Chillaregen mukaan valituilla tyyppiluokilla tulee ensinnäkin olla riittävästi ulottuvuuksia kattamaan koko prosessi, josta ollaan kiinnostuneita. Virhetyyppien tulee olla sellaisia, että niiden jakauma muuttuu tuotteen siirtyessä prosessin vaiheesta toiseen, jotta luokittelun edellytys, jakauman käyttö mittarina, on ylipäätään mahdollista. Luokkien tulee olla mahdollisimman erillisiä ja riippumattomia toisistaan sekä niiden tulee olla sovellettavissa kaikissa prosessin vaiheissa. Toisin sanoen niiden on oltava riittävän geneerisiä käytettäväksi kaikissa aktiviteeteissa [Chi94].

Luokittelua seuraa koulutus, jossa luokittelijat perehdytetään luokittelun käyttöön. Näin varmistetaan, että saatu data sisältää mahdollisimman vähän virheitä. Luokittelun tuloksena saadut tiedot tulee tallentaa helposti saatavilla olevaan tietokantaan. Mikäli yrityksellä on jo käytössä on väline, jolla kerätään virheisiin liittyvää informaatiota, se kannattaa muokata soveltuvaksi myös ODC-luokittelun tarvitsemien tietojen tallennukseen. Luokiteltuja virheitä tulee säännöllisesti validoida, jotta voidaan varmistua luokituksen johdonmukaisuudesta ja oikeellisuudesta. Tietojen validoinnin voi suorittaa joko tarvittavat taidot omaava henkilö, tai siihen voidaan käyttää aggregaattianalyysia. Datan oikeellisuuden varmistaminen on erityisen tärkeää

juuri luokittelun alussa, jolloin luokittelun suorittajilla voi vielä olla epävarmuutta peruskäsitteistä ja luokkien käytöstä. Datan laatu ei yleensä enää myöhemmissä vaiheissa ole ongelma [BMK02].

Seuraavassa vaiheessa saatua dataa analysoidaan. Analysoinnin suorittajaksi valitaan tekninen henkilö, joka tuntee projektin ja jolla on tiedon arviointiin tarvittavat taidot. Avuksi tarvitaan helppokäyttöinen datan visualisointityökalu, jolla tarvittavat virhetyyppi- ja laukaisinprofiilit on helppo muodostaa [BMK02].

Mikäli historiallista virhedataa on saatavilla eikä prosessi muutu, on varsin helppoa muodostaa virhetyyppien tai –laukaisinten oletusprofiileja, joita tarvitaan virheiden sekä niihin liittyvien riskien tunnistamiseen meneillään olevassa projektissa. Mikäli vertailudataa ei ole saatavilla, tutkijat Basin, Kratschmer ja Santhanam suosittavat odotusprofiilin muodostamista seuraavasti:

> Käytettävä prosessimalli (vesiputous, iteratiivinen, spiraali tai muu) identifioidaan, jotta dataa on helpompi organisoida ja tulkita. Prosessin eri aktiviteetit tunnistetaan, nimetään ja kunkin vastuut määritellään lyhyesti. Aktiviteetit kytketään niihin laukaisimiin, joista kyseinen aktiviteetti vastaa. Tieto toimii perustana odotetun virhetyyppiprofiilin (defect type signature) muodostamisessa.
>
> Kuhunkin aktiviteettiin liitetään painoarvo sille varattujen resurssien mukaisessa suhteessa koko prosessiin. Lopuksi historiallisesta datasta saatavaa tietoa laukaisinten suhteesta virhetyyppeihin käytetään oletetun virheprofiilin muodostamiseen kullekin aktiviteetille [BKS98].

Analysointivaiheessa muodostuneita virheprofiileja verrataan joko aikaisempiin vastaaviin tai oletusprofiileihin. Olennaisia ovat kokonaisdatasta muodostuvat trendit ja muodot, eivät yksittäisiin virheisiin liittyvät tiedot [BMK02]. Analysoinnista saatava tieto välitetään säännöllisesti projektiryhmälle sekä validoinnista vastaaville yksiköille. Näin parannetaan itse luokituksen laatua, mahdollistetaan tarvittavat parannustoimenpiteet prosessin aikana sekä sitoutetaan ryhmät paremmin ODC-luokittelun

käyttöön. Kun ryhmät ovat saaneet palautteen, ne voivat sisäisesti tunnistaa ja priorisoida mahdollisesti tarvittavat korjaustoimenpiteet [BMK02].

Vasta kun ODC-luokittelu on saatu integroiduksi osaksi ohjelmistotuotantoprosessia, sen maksimaalinen hyödyntäminen mahdollistuu. Luokittelun avulla voidaan jatkuvasti seurata kehitysprosessia ja sen aikana tapahtuvia tuotteen luotettavuuden muutoksia. Tällöin tuotteen laadun rakentaminen jo aivan prosessin alusta lähtien voidaan varmistaa.

4 Menetelmät

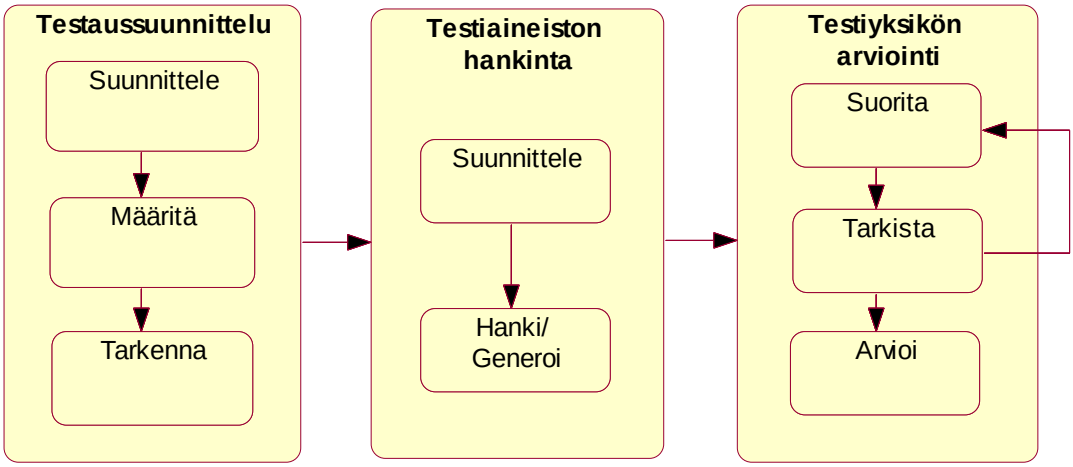
Sovellettavista menetelmistä päättäminen on osa ohjelmistotuotantoprosessin hallintaa. Ensimmäisenä toimenpiteenä yksikkö- ja integrointitestauksen kehittämisessä on näiden vaiheiden strukturointi testaustehtäviin. Kukin tehtävä ohjeistetaan selkeästi ja kussakin tehtävässä sovellettavat menetelmät nimetään. Myös eri vaiheissa vaadittava dokumentointi määritellään, sillä dokumentointi on tärkeä osa yrityksen sisäistä ja ulkoista viestintää ja liiketoiminnan jatkuvuuden edellytys.

4.1 Testaustehtävät ja -menetelmät

Yksikkö- ja integrointitestauksen jäsentämisen ja kehittämisen ensimmäinen edellytys on näihin testausvaiheisiin liittyvien testaustehtävien ja dokumentaation määrittely, ohjeistus, kokeilu projekteissa ja käyttöönotto. Lähtökohtana testaustehtävien ja dokumentaation määrittelylle kannattaa käyttää IEEE:n määrittelemiä standardeja, joihin on koottu suuri määrä alan asiantuntemusta. Valmiin standardin pohjalta yrityksen on helpompi ottaa jo alusta alkaen huomioon kaikki tarvittavat seikat, eikä resursseja tarvitse käyttää pitkälliseen yrityksen ja erehdyksen kautta oppimiseen. Standardit eivät kuitenkaan ole mikään suoraan käyttöön valmis käytäntö, vaan vaativat yritykseltä sopeuttamisen yrityksen omaan ohjelmistotuotannon prosessiin, malleihin ja käytäntöihin.

IEEE:n standardi 1008 yksikkötestaukselle (IEEE Standard for Software Unit Testing) pyrkii tarjoamaan lähestymistavan systemaattiseen ja dokumentoituun yksik-

kötestaukseen. Yksikkötestaus jaetaan kolmeen vaiheeseen sekä näiden vaiheiden sisältämiin toimenpiteisiin (activities), joita on yhteensä kahdeksan. Kuva 6 esittää standardin 1008 vaihejaon ja kuhunkin vaiheeseen liittyvät toimenpiteet.



Kuva 6 IEEE1008 standardin määrittelemät yksikkötestauksen vaiheet ja toimenpiteet

Ensimmäisen testaussuunnitteluvaiheen ensimmäisenä toimenpiteenä suunnitellaan yleinen lähestymistapa yksikkötestaukseen: tunnistetaan testauksessa huomioitavat riskialueet, selvitetään saatavilla olevat syöte- ja tulostetiedot sekä miten tämä data voidaan validoida ja miten tulostiedot kerätään, talletetaan ja validoidaan. Lisäksi päätetään testauskriteeri, joka määrittelee, koska testauksen tavoite on saavutettu ja testaus voidaan lopettaa. Varsinaisen testauskriteerin lisäksi määritellään myös ne erikoistilanteet, joissa testaus keskeytetään ennen varsinaisen testauskriteerin täyttymistä ja millaista ilmoitusmenettelyä näissä tilanteissa sovelletaan. Testaukseen tarvittavat resurssit määritellään. Samoin määritellään yleinen aikataulu resurssien ja testattavien yksiköiden saatavuuden perusteella. Mikäli testaus käsittää useamman kuin yhden yksikön testauksen, yleinen lähestymistapa suunnitellaan kerralla kaikille testattaville yksiköille. Muut testaussuunnitteluvaiheen toimenpiteet tulee standardin mukaan suorittaa vähintään kertaalleen kullekin testattavalle yksikölle [IEEE1008].

Seuraavaksi määritellään testattavat piirteet. Toiminnalliset määrittelyt käydään läpi ja täydennetään ei-toiminnallisin vaatimuksin. Yksikön tilat ja luvalliset tilasiirtymät tunnistetaan. Syötteiden ja tulosteiden ominaisuudet käydään läpi ja valitaan testaukseen sisällytettävät piirteet. Koska yksiköiden täydellinen testaus on kallista ja epäkäytännöllistä, yksikön todennäköisestä käyttötavasta saatavaa informaatio tulee

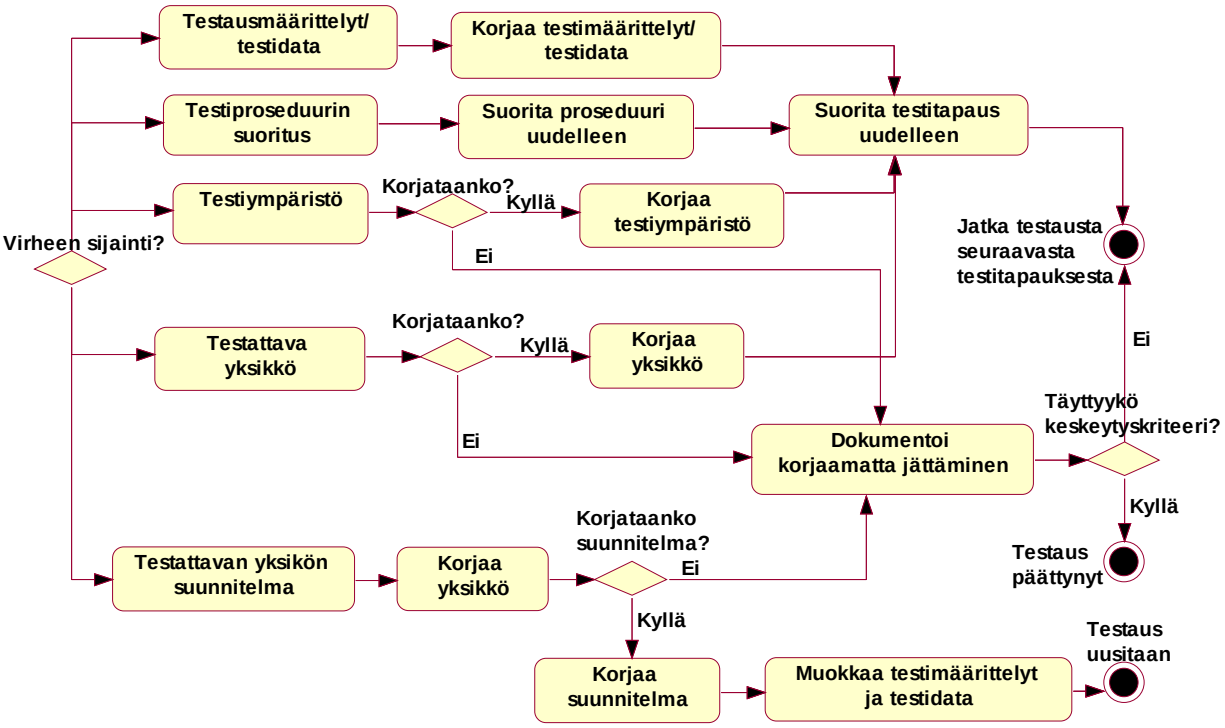
hyödyntää valittaessa testattavia piirteitä. Testaamattomiin osiin liittyvät riskit tulee tunnistaa. Viimeisenä suunnitteluvaiheessa tarkennetaan laadittua yleissuunnitelmaa. Mahdollisesti saatavilla olevat valmiit testitapaukset ja -proseduurit selvitetään. Tarvittavat erityisresurssit tunnistetaan ja varataan. Myös aikataulua tarkennetaan [IEEE1008].

Seuraavan vaiheen alussa suunnitellaan testijoukon arkkitehtuuri. Testattavien piirteiden ja aiemmin määriteltyjen tekijöiden (tilasiirtymät, datan luonne ym.) perusteella laaditaan hierarkkisesti rakentuva testitavoitteiden joukko. Tavoitteena on, että alimman tason testaustavoitteet voidaan suoraan testata muutamalla testitapauksella. Lisäksi hankitaan tarvittavat testiproseduurit sekä testitapausmäärittelyt, joita täydennetään tehdyn testitavoitteiden arkkitehtuurisuunnitelman perusteella. Testaussuunnitelma (test design specification) täydennetään valmiiksi [IEEE1008]. Yksikkötestausstandardi suosittaa soveltuvin osin IEEE:n dokumentointistandardin (IEEE-standardin 829-1983) mukaisen dokumentoinnin käyttöä myös yksikkötestauksessa. Käsittelen dokumentointistandardin sisältöä ja soveltamista lähemmin luvussa 4.2.

Seuraavaksi tarvittava testiaineisto hankitaan ja verifioidaan tai tarvittaessa luodaan. Hankitaan tarvittavat erityisresurssit, testauksessa tarvittavat manuaalit, käyttöjärjestelmäproseduurit, ohjaustiedot (taulut ym.) sekä sellaiset ohjelmistot, joilla on rajapinta testattavaan yksikköön [IEEE1008].

Viimeisessä vaiheessa testit suoritetaan ja niiden tulokset arvioidaan. Löydetyt virheet analysoidaan. Kuva 7 havainnollistaa löydetyn virheen käsittelyä. Mikäli virheen syy on testausmäärittelyissä tai testidatassa, virhe korjataan ja epäonnistunut testitapaus suoritetaan uudelleen. Jos virhe tapahtui testiproseduurin suorituksessa, virheellisesti suoritettu proseduuri ajetaan uudelleen. Mikäli virheen syyksi paljastuu testiympäristö, virhe voidaan korjata ja testit uusia. Joissakin tapauksissa virhe voidaan kuitenkin päättää jättää korjaamatta, jolloin selvitys ympäristön korjaamatta jättämisestä liitetään testiyhteenvetoraporttiin. Jos näin menetellään, seuraavaksi tutkitaan täyttyykö jokin testauksen keskeytyskriteereistä. Jos edellytys testauksen keskeyttämiselle on olemassa, testaus lopetetaan. Mikäli virhe löytyy testattavasta yksiköstä, se korjataan ja testit ajetaan uudelleen, tai korjaamattajättämispäätös kirjataan yhteenvetoraporttiin ja tarkistetaan, edellyttääkö tilanne testauksen keskeyttämistä.

Virhe voi olla myös yksikön suunnitelmassa, jolloin suunnitelma ja yksikkö korjataan, testimäärittelyt ja data muokataan yksikön testaamiseen sopiviksi ja testit uusitaan tai päätetään jättää suunnitelma korjaamatta, jolloin päätös kirjataan yhteenvetoraporttiin ja selvitetään testauksen keskeytyskriteerien täyttyminen [IEEE1008].



Kuva 7 Havaitun virheen käsittely IEEE-standardin 1008 mukaan

Testitapausten suoritusta jatketaan, kunnes testauksen päätöskriteeri, joka määriteltiin ensimmäisessä vaiheessa, täyttyy. Mikäli kaikki testitapaukset on suoritettu, mutta testauksen päätöskriteeri ei täyty, testitapausten joukkoa täydennetään [IEEE1008].

Viimeisenä testiyhteenvetolomakkeelle kirjataan poikkeamat testisuunnitelmasta, testimäärittelyistä sekä mahdollisesta testauksen keskeytyksestä. Mikäli testaus on keskeytetty jonkin poikkeusehdon nojalla, riittämättömästi testatut alueet merkitään myös lomakkeelle. Erot testattavassa yksikössä ja sen määrittelyissä kirjataan ylös. Testitulosten ja löydettyjen virheiden perusteella arvioidaan yksikön arkkitehtuuria ja sen implementointia suhteessa vaatimuksiin. Tulos kirjataan yhteenvetolomakkeelle. Mikäli IEEE-standardin 829 mukaista yhteenvetolomaketta käytetään, sitä täyden-

netään myös muilta osin. Lopuksi käytetty testiaineisto – testisuunnitelma, testitapaukset, testiproseduurit, testidata, ajurit, tyngät ja yhteenvetoraportti - kerätään, organisoidaan ja talletetaan uudelleenkäyttöä varten [IEEE1008].

Käytännön ohjelmistotuotannon käyttöön standardin tarjoama malli on liian raskas sovellettavaksi jokaisen yksikön testaamiseen. Testattavat yksiköt ovat testauksen varhaisvaiheissa hyvin pieniä ja niitä on paljon. Ei-turvallisuuskriittisissä ohjelmissa tuotantokustannukset nousevat kohtuuttomiksi liian perusteellisella yksikkö- ja integrointitestauksella. Mallin tarjoama runko testausprosessin mallintamiseen on kuitenkin itsessään käyttökelpoinen sekä yksikkö- että integrointitestaukseen.

Tehtävien ja tuotettavan dokumentaation tarkkuus ja määrä kannattaa yrityksen kuitenkin määritellä omien luotettavuus- ja kustannustavoitteidensa perusteella. Apuna yritys voi käyttää IEEE-standardia 1012 ohjelman verifioinnille ja validoinnille (IEEE Standard for Software Varification and Validation), joka määrittelee ohjelman eri osille suoritettavat vähimmäisverifiointi- ja validointitehtävät testattavan yksikön kriittisyyden perusteella (Taulukko 5).

Kriittisyys	**Kuvaus**
Kriittinen	Valittu toiminto vaikuttaa kriittisesti järjestelmän suorituskykyyn
Merkittävä	Valittu toiminto vaikuttaa merkittävästi järjestelmän suorituskykyyn
Kohtalainen	Valittu toiminto vaikuttaa järjestelmän suorituskykyyn, mutta erilaisia ongelman kiertävillä menettelyillä voidaan korvata suorituskyvyn menetys
Matala	Valitulla toiminnolla on havaittava vaikutus ohjelman suorituskykyyn, mutta asiakas kokee tilanteen ainoastaan epämukavana, mikäli toiminto ei toimi määritysten mukaan

Taulukko 5 Verifioitavan tai validoitavan yksikön kriittisyys [IEEE1012]

Standardin 1012 jaottelua testattavan osan kriittisyyden mukaan voi soveltaa rinnan standardin 1008 yksikkötestausohjeistuksen kanssa. Toiminnaltaan kriittisille tai laajaan uudelleenkäyttöön suunnitelluille yksiköille voidaan yrityksessä sopia sovellettavaksi tarkempaa menettelyä yksikkötestaus- ja integrointitestausvaiheissa, kun taas vähemmän kriittisten osien testaukseen kannattaa soveltaa hyvin kevyttä, mutta tavoitteisiin nähden mahdollisimman tehokasta testausta.

Yksikkötestauksen vaatimien ylimääräisten ohjelmakomponenttien (ajurit ja tyngät) määrän minimoimiseksi on tärkeää aikatauluttaa ohjelman tuottaminen ja yksikkö- ja integrointitestaus sellaiseksi, että näitä komponentteja tarvitaan mahdollisimman vähän. Eräs käyttökelpoinen menetelmä on Beizerin suosittama menettely, jossa ensin tuotetaan, testataan ja integroidaan ohjelman ydinohjelma (backbone), jota huolellisen testauksen jälkeen käytetään ensisijaisena testausalustana. Tynkien osalta on myös huomattava, että tynkäprosessi on aina simulaatio, josta voi aiheutua kokonaan uusi joukko virheitä. Lisäksi tynkä itsessään on toimiva yksikkö ja tulisi siis myös yksikkötestata [Bei84].

Integrointitestaukseen ei kirjallisuudessa ole yksikkötestauksen IEEE-standardia 1008 vastaavia standardeja. Integrointitestauksen tehtävät ovat käytännössä kuitenkin samat kuin yksikkötestauksessa sovellettavat. Olennainen ero testausvaiheissa on testauksen tavoitteessa. Yksikkötestauksessa pyritään todentamaan, että testauksen kohteena oleva yksikkö toimii oikein [JoE94]. Integrointitestauksessa yksiköt yhdistetään toisiinsa ja testataan niiden välistä yhteistyötä, kommunikointia ja rajapintoja [Bei90]. Rajapinnoissa olevat virheet eivät paljastu yksikkötestauksen aikana. Integrointitestauksen avuksi Myers esittää teoksessaan The Art of Software Testing [Mye79] tarkistuslistaa, jota voi hyödyntää rajapintojen testauksen suunnittelussa. Valitettavasti lista soveltuu käytettäväksi ainoastaan proseduraalisilla ohjelmointikielillä laadittujen ohjelmien integrointitestauksen apuvälineenä.

Alkuvaiheessa, kun yritys on vasta kehittämässä omia yksikkö- ja integrointitestauksen menetelmiään ja standardejaan, kannattaa aloittaa varsin kevyellä testaustehtävien määrittelyllä. Testattavat osat voidaan joko luokitella niiden kriittisyyden perusteella ja ohjeistaa eri kriittisyysasteilla olevat osiot testattaviksi eri tavoin tai aloittaa yhdellä, kaikkiin tilanteisiin soveltuvalla, mutta riittävän kevyellä testitehtävien ohjeistuksella, jota voidaan laajentaa ja täsmentää kun ensimmäinen versio on saatu integroitua osaksi normaalia sovelluskehitysprosessia. Standardin 1008 tarjoama kehys yksikkötestaukselle on hyvä lähtökohta ja muodostaa selkeän rungon testauksen eri vaiheille ja tehtäville, mutta on suoraan käytäntöön sovellettuna liian vaativa ja raskas menettely.

Liian raskas yksikkö- ja integrointitestauksen ohjeistus, tehtävämäärittelyt ja dokumentointi kasvattavat merkittävästi ohjelmistotuotannon kustannuksia tarjoamatta vastineeksi selkeää parannusta ohjelman luotettavuudessa. Lisäksi raskaisiin ohjeistuksiin ja dokumentointeihin sisältyy vaara, ettei toimintatapa koskaan integroidu osaksi tuotteen kehitysprosessia, vaan jää hiljalleen pois käytöstä. Mikäli sen käyttöä johdon taholta vaaditaan, tehtävät ja dokumentointi saatetaan hoitaa tavalla, joka ei lisää ohjelmaan laatua, mutta juuri ja juuri täyttää ohjeistuksen kirjaimelliset määräykset.

4.2 Testauksen dokumentointi

IEEE:n standardi 829 määrittelee kahdeksan erilaista perusdokumenttia, joita voidaan soveltaa testauksen eri vaiheiden suunnittelussa, määrittelyssä ja raportoinnissa. Standardin tarjoama dokumentaatio on sovellettavissa kaikille testauksen tasoille yksikkötestauksesta hyväksymistestaukseen saakka. Standardi ei kuitenkaan määrittele, mitä dokumentteja tulee missäkin testauksen vaiheessa tuottaa. Organisaation tuleekin standardin mukaisen dokumentaation käyttöönottoa harkittaessa päättää, mihin ohjelmatyyppeihin standardia sovelletaan ja mitä dokumentteja kullakin testauksen tasolla vaaditaan. Myös standardin tarjoamien dokumenttien sisältöä tulee muokata ohjelmatyypin ja testauksen vaiheen mukaan yrityksen käyttöön sopiviksi [IEEE829].

Testaussuunnitelma (test plan) määrittelee testauksen puitteet, lähestymistavan, resurssit sekä testausaktiviteettien aikataulun. Se identifioi testattavat yksiköt, piirteet, suoritettavat testaustehtävät, näistä vastaavat henkilöt ja kyseiseen suunnitelmaan sisältyvät riskit [IEEE829].

Testauksen suunnittelua seuraa määrittelyvaihe, johon standardi 829 tarjoaa kolme dokumenttia. Testisuunnitelman täsmennys (test design spesification) tarkentaa testauksen lähestymistapaa sekä identifioi ne piirteet, jotka tällä suunnitelmalla ja siihen liittyvillä testitapauksilla ja -menettelyillä testataan. Testitapausmäärittely (test case specification) määrittää suoritettavan yksittäisen testitapauksen: testattavan yksikön, sen syöte- ja tulostiedot sekä testiympäristön (laitteet ja ohjelmistot). Lisäksi testita-

pausmäärittelyyn dokumentoidaan yksikön testaamiseen vaadittavat erityismenettelyt ja kyseisen testitapauksen kytkennät muihin testitapauksiin, esimerkiksi tieto siitä, mitkä testitapaukset tulee suorittaa ennen tätä. Testausmenettelyn selostus (test procedure specification) erittelee testitapausten joukon suoritusvaiheet tai yleisemmin ohjelmayksikön analysoinnin vaiheet [IEEE829].

Testauksen raportointiin on tarjolla neljä dokumenttia. Testaukseen toimitusraportti (test item transmittal report) sisältää listan testattavaksi toimitetuista kohteista. Testiloki (test log) on kronologinen nauhoite relevanteista testin suoristusaikaisista yksityiskohdista. Testitapausraportti (test incident report) dokumentoi testausprosessin aikana ilmenneen ja tutkimusta vaativan tapahtuman ja testausyhteenvetoraportti (test summary report) sisältää yhteenvedon suunniteltujen testaustoimenpiteiden suorituksesta, testauksen kulusta sekä tuloksiin perustuvan arvioinnin testatuista kohteista [IEEE829].

IEEE:n dokumentointistandardin käyttöönotossa kehotetaan ensimmäisessä vaiheessa ottamaan käyttöön suunnittelu- ja raportointidokumentaatio soveltuvin osin. Testaussuunnitelma luo pohjan koko testausprosessille, raportointidokumentit puolestaan auttavat testauksessa saadun olennaisen informaation tallentamista organisoidulla tavalla [IEEE829].

Yksikkötestauksessa dokumentointistandardi suosittaa dokumentoinnin hoitamista siten, että yksi dokumentti, esimerkiksi testaussuunnitelma, kattaa useita testattavia yksiköitä. Jokaisesta testattavasta yksiköstä ei ole järkevää tehdä omaa dokumentointia. Yksikkötestausvaiheessa käytettäviksi dokumenteiksi standardi suosittaa testisuunnitelman täsmennyksen, testitapausmäärittelyjä sekä testauksen yhteenvetoraporttia [IEEE829].

Kaner, Bach ja Pettichord [KBP02] kritisoivat voimakkaasti standardia 829. Heidän mukaansa standardi olettaa käytettäväksi vesiputousmallin mukaista tuotekehitystä, jossa testaus suunnitellaan aikaisessa vaiheessa, dokumentoidaan huolella, eikä sitä sen jälkeen muuteta. Kuitenkin käytännön projekteissa aina ilmenevät muutostarpeet tekevät luodun dokumentaation ylläpidosta erittäin kallista. Heidän mukaansa standardin yhteydessä ei useinkaan tiedosteta tai keskustella suuren dokumenttimäärän

tuottamisen ja ylläpidon kustannuksista. Dokumentointiin käytetty aika on aina poissa henkilön varsinaisesta lisäarvoa tuottavasta työstä. Lisäksi standardin käyttö heidän mukaansa korostaa dokumentaation määrää sen laadun kustannuksella. Hyveeksi nousee mahdollisimman suuren dokumentaation tuottaminen kyseenalaistamatta sitä, tarvitaanko kyseistä dokumentaatiota. Suuri dokumentaatio myös kätkee helposti virheitä. Esimerkiksi olennaisia testitapauksia on voinut jäädä pois, mutta asian todentaminen on hankalaa dokumentaation käsittäessä satoja sivuja. Standardin soveltamisen sijaan tutkijat suosittavatkin käyttämään vähemmän formaalia dokumentaatiota: lyhyiden listojen ja taulukoiden kokoelmia, tilaraportteja, huolellisia virheraportteja. Tarvittava kommunikointi voidaan hoitaa myös säännöllisillä ryhmäkokoontumisilla.

Yrityksen tuleekin huolella harkita, millaista dokumentaatiota päätetään soveltaa yksikkö- ja integrointitestausvaiheissa. Lisäksi dokumentoinnin muoto ja tarkkuus kannattaa suunnitella sellaiseksi, että dokumentaation tuottamisen ja ylläpidon aiheuttamat kustannukset eivät nouse kohtuuttomiksi dokumentoinnin tuottamaan hyötyyn nähden. Dokumentointi ei ole arvo sinänsä, vaan sen määrä ja laatu on mitoitettava tarpeen mukaisiksi.

Yrityksessä käytettäviä mittareita varten voidaan tarvita myös tietoa näissä vaiheissa löydettyjen virheiden määristä. Myös virheraportoinnin ja virheisiin liittyvän dokumentointivelvoitteen tulee olla prosessin vaiheeseen nähden soveltuva. Liian raskas virheraportointimenettely testauksen alkuvaiheessa, jossa virheitä tavoitellaan löytyväksi mahdollisimman paljon, voi liiaksi siirtää resursseja varsinaisesta ohjelman tuottamisesta virheiden kirjaamiseen ja raportointiin ja näin aiheuttaa huomattavia kustannuksia vähentyneenä tuottavuutena [Bei84]. Virheraporttien tuottamisen aiheuttamien kustannusten lisäksi niiden käsittely organisaatiossa aiheuttaa kustannuksia, jotka tulee huomioida virheiden raportointia suunniteltaessa ja ohjeistettaessa [KBP02].

5 Yksikkö- ja integrointitestauksen mittarit

Ohjelmistotuotannossa tarvitaan mittaustietoa, jotta ymmärretään, mitä ohjelman kehityksen ja ylläpidon aikana tapahtuu. Lisäksi mittarit mahdollistavat projektien kontrolloinnin ja sitä kautta prosessin ja myös tuotteen parantamisen [FeP96]. Yksikkö- ja integrointitestausvaiheiden osalta keskeisiä tietoja ovat suoritetun testauksen riittävyys sekä tuotteen ja prosessin laatu, jota yleensä alkuvaiheessa joudutaan arvioimaan erilaisilla virheiden määrästä johdetuilla suureilla. Erilaisia virheiden määristä johdettuja mittareita on suuri määrä [Kan00]. Näistä tutkielmassani esittelen vain muutaman yleisimmin käytetyn.

5.1 Rakenteellisen testauksen mittarit

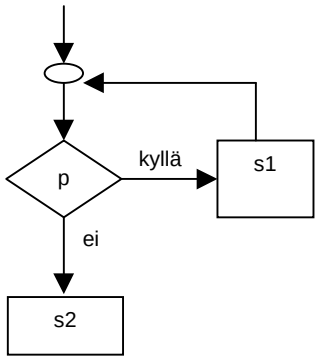
Eräs keskeisiä ongelma testauksessa on sen määrittäminen, milloin ohjelmaa on testattu riittävästi. Toisinaan riittävyys määritellään hyvin epämuodollisesti: testaus lopetetaan, kun prosessissa testaukseen varattu aika tai testausbudjetti on käytetty [Hum89]. Testaus voidaan myös katsoa riittävästi suoritetuksi, kun ohjelma on hyväksyttävästi suorittanut kaikki testitapaukset. Testitapausten lukumäärä tai testausbudjetin suuruus ei kuitenkaan kerro mitään testauksen tehokkuudesta tai laadusta. Sen vuoksi tarvitaan mieluiten numeerinen, selkeä, yksiselitteinen ja todennettavissa oleva kriteeri, jonka avulla voidaan varmistua testauksen riittävyydestä ja päättää ohjelman siirtämisestä ohjelmistokehitysprosessin seuraavaan vaiheeseen.

5.1.1 Polkutestaus

Suoritetun rakenteellisen testauksen määrän mittarina käytetään usein erilaisia kattavuuksia(coverage). Kattavuus on luku, joka kertoo, kuinka suuren osan ohjelmasta testitapausten joukko valitun kattavuusstrategian perusteella suorittaa. Kattavuuksien laskeminen perustuu ohjelman kontrolli- tai tietorakenteiden muodostamiin suorituspolkuihin, joita usein havainnollistetaan lohkokaavioina.

Lohkokaavio (Kuva 8) on yksinkertaistettu graafinen malli ohjelman kontrollirakenteesta, ja sitä käytetään ohjelman staattiseen analysointiin. Kyseessä on suunnattu verkko, joka kostuu joukosta solmuja (node) ja näitä yhdistäviä kaaria (edge). Kukin

kaari edustaa kontrollin siirtymistä ohjelman sisällä ja muodostuu järjestetystä parista solmuja. Lisäksi lohkokaaviossa on alkusolmu eli solmu, johon ei tule yhtään kaarta, sekä loppusolmu, josta puolestaan ei lähde yhtään kaarta. Verkon jokaisen solmun tulee kuulua johonkin alkusolmusta alkavaan ja loppusolmuun päättyvään polkuun [ZHM97]. Solmut voivat olla joko ohjelman suoritettavia lauseita (statement) tai ehto- eli predikaattisolmuja, joissa ohjelman kulku haarautuu eri polkuihin predikaatin määrittämän ehdon perusteella. Kaaret yhdistävät solmuja toisiinsa ja määrittävät ohjelman kulun solmujen välillä. Polku on ohjelman alkusolmusta alkava sarja kaarien toisiinsa yhdistämiä solmuja, joka kulkee halki ohjelman ja päättyy loppusolmuun.



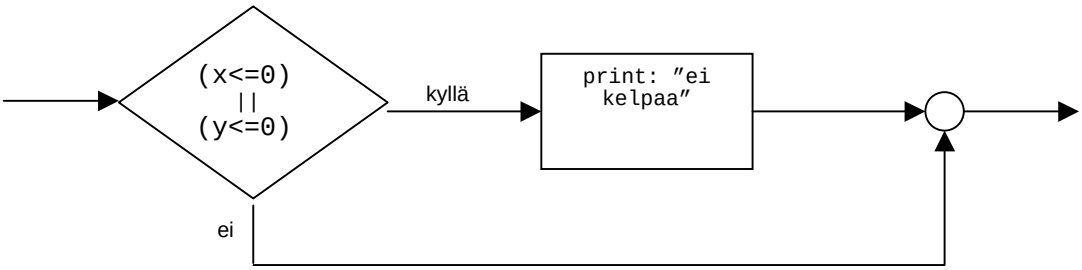
Kuva 8 **Lohkokaavio. Ehto-/predikaattisolmu p sekä suoritettavat lauseet s1 ja s2, nuolet kaaria**

Lohkokaavion muodostaman rakenteellisen mallin perusteella muodostetaan testitapaukset, jotka suorittavat halutun kattavuuden saavuttamiseksi tarvittavat suorituspolut. Polkutestaus on lasilaatikkotestaustekniikka ja soveltuu parhaiten juuri yksikkötestausvaiheen riittävyyden määrittämiseen, sillä polkutestauksen teho heikkenee nopeasti testattavan yksikön koon kasvaessa [Bei90, Mar91]. Pyrkimyksenä on saavuttaa mahdollisimman korkea kattavuus mahdollisimman pienellä testitapausten joukolla. Syynä tähän ovat paitsi kustannukset myös tehokkuus. Monimutkaiset testitapaukset ovat yleensä ohjelmallisesti haastavampia käsitellä ja paljastavat näin todennäköisemmin virheitä [Mar91].

Lausekattavuus (statement coverage) on eräs hyvin tavallinen ohjelmalle suoritetun testauksen määrän mittari. Ohjelma on saavuttanut 100 % lausekattavuuden, jos jokainen ohjelman lause suoritetaan vähintään kerran. Tällöin myös jokainen kontrol-

lilohkokaavion solmu kuuluu johonkin testitapausten joukon suorittamaan polkuun [ZHM97]. Lausekattavuus on IEEE:n määrittelemän yksikkötestausstandardin [IEEE 1008] vaatima testauksen kattavuusminimi sekä myös IBM:n yhtiöstandardi jo 30 vuoden ajalta. Täyden lausekattavuudenkaan saavuttava testitapausten joukko ei kuitenkaan käy läpi kaikkia ohjelman kontrollin haarautumisia kontrollirakenteissa [ZHM97]. Olio-ohjelmissa lausekattavuus ei puolestaan huomioi yksikön sisäisen tilan vaikutusta lauseen suoritukseen [Bin00]. Lausekattavuus onkin hyvin heikko kattavuuskriteeri ja käsitettävä lähinnä testauksen riittävyyden arvioinnin minimivaatimukseksi [Bin00]. Lohkokaavioesimerkissä (Kuva 9) 100 %:n lausekattavuus saavutetaan yhdellä testitapauksella, esimerkiksi tapauksella $x = 0$.

Suurissa järjestelmissä 100 %:n lausekattavuutta on usein mahdoton saavuttaa. Syynä tähän ovat polut, joita ei koskaan todellisuudessa ole mahdollista suorittaa, niin sanottu ”kuollut koodi” (dead code) [ZHM97]. Valmiissa ohjelmassa tällaista kuollutta koodia katsotaan tyypillisesti olevan noin 10-15 % [Bin00].



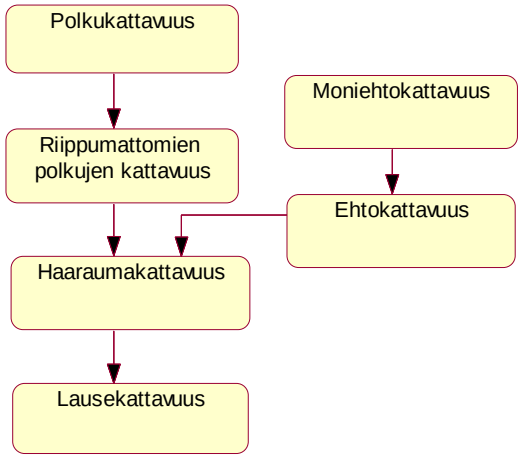
Kuva 9 Lohkokaavioesimerkki

Täyden **haaraumakattavuuden** (brach coverage) saavuttamiseksi testitapausten joukon tulee olla sellainen, että jokainen kontrollilohkokaavion kaari kuuluu johonkin testitapausten joukon suorittamaan polkuun. Käytännössä tämä tarkoittaa, että kukin ohjelman ehto- eli predikaattisolmu saa testitapausten joukolla vähintään kertaalleen arvon tosi ja vähintään kertaalleen arvon epätosi. Haaraumakattavuus sisältää lausekattavuuden, eli jokainen testitapausten joukko, joka saavuttaa täyden haaraumakattavuuden, saavuttaa aina myös täyden lausekattavuuden. Kuva 10 esittää tutkielmassa mainittujen kattavuuksien välisen hierarkkisen järjestyksen. Kaikkien haarautumien suorittaminen ei kuitenkaan tarkoita, että kaikki mahdolliset kontrolli-

siirtymien yhdistelmät olisi testattu [ZHM97]. Lohkokaavioesimerkissä (Kuva 9) 100 %:n haaraumakattavuus vaatii kaksi testitapausta: *x = 0* (tosi) ja *x = 1* (epätosi) .

Kontrollin haarautumien testaamiseen keskittyvät ehto- ja moniehtokattavuus. **Ehtokattavuus** (condition coverage) edellyttää, että kunkin predikaattisolmun atomisen predikaatin (yksittäisen, jakamattoman ehtolauseen) tulee testitapausten joukolla saada vähintään kerran arvo tosi ja kerran epätosi. Lohkokaavioesimerkissä (Kuva 9) 100 %:n ehtokattavuuteen tarvitaan neljä testitapausta: *x = 0* (x tosi) , *x = 1* (x epätosi) , *y = 0* (y tosi) ja *y =1* (y epätosi). Ehtokattavuus sisältyy **moniehtokattavuuteen** (multiple condition coverage), jonka saavuttaakseen testitapausten joukon tulee kattaa kaikki mahdolliset atomisten predikaattien totuusarvojen yhdistelmät jokaisessa ehtosolmussa [ZHM97]. Esimerkkitapauksessa (Kuva 9) 100 %:n moniehtokattavuus edellyttää myös neljää testitapausta, mutta toisin kuin ehtokattavuuden testitapauksissa, on jokaisessa testitapauksessa asetettava molempien muuttujien arvot: *x = 0 ja y = 0* (x tosi, y tosi), *x = 0 ja y = 1* (x tosi, y epätosi), *x = 1 ja y = 0* (x epätosi, y tosi), *x =1 ja y =1* (x epätosi, y epätosi).

Täydellinen **polkukattavuus** (path coverage) saavutetaan testitapausten joukolla, joka suorittaa kaikki ohjelman polut alkusolmusta loppusolmuun. Käytännössä polkukattavuutta on lähes mahdoton saavuttaa, sillä silmukat lisäävät mahdollisten suorituspolkujen määrän nopeasti valtavan suureksi [ZHM97] eikä se siksi ole käytännön ohjelmistotuotannossa käyttökelpoinen mittari. Käyttökelpoisempi on **riippumattomien polkujen kattavuus** (independent path coverage). Siinä rajoitetaan testattavien polkujen määrä koskemaan ainoastaan polkuja, jotka ovat muista riippumattomia, toisin sanoen joita ei voi muodostaa lineaarisena kombinaationa muista testatuista poluista [ZHM97]. Riippumattomien polkujen määrä ohjelmassa on predikaattisolmujen määrä lisättynä yhdellä [Pre00].



Kuva 10 Kattavuushierarkia

Kaikkien tietovuohon perustuvien kattavuuskriteerien tavoitemäärittelyssä on tärkeää huomioida, että ohjelmakoodissa varsin todennäköisesti on sellaisia rakenteita (lauseita, ehtoja, polkuja), joita ei ole mahdollista suorittaa millään testitapauksella. Sen takia käytännössä usein asetetaan tietty kattavuustavoite koskemaan vain testattavissa olevat tapaukset. Toisinaan on kuitenkin mahdotonta selvittää, onko jokin ohjelman kohta suoritettavissa jollakin testitapauksella [ZHM97]. Käytännön tasolla tämä tarkoittaa, ettei täydellisen kattavuustavoitteen asettaminen useinkaan ole edes mahdollista saati resurssien käytön kannalta perusteltavissa. Su & Ritter kehottavat esimerkiksi haaraumakattavuuden osalta olettamaan suoraan 15 % ohjelmakoodista sellaiseksi, ettei sitä voida saavuttaa millään testitapauksella (kuollut koodi), jolloin tavoiteltavaksi haaraumakattavuudeksi testitapausten joukolla jää 85 % [SuR91]. Näin vältytään kalliilta ja aikaa vieviltä selvittelyiltä, onko jokin yksittäinen ohjelman suorituspolku mahdollista saavuttaa vai ei.

5.1.2 Tietovuotestaus

Tietovuotestauksessa (data flow testing) testattavat polut valitaan ohjelman muuttujien määrittelyjen (value assigment) ja käyttöjen (value uses) sijainnin perusteella. Kustakin analysoitavasta muuttujasta tuotetaan oma tietovuokaavio, jonka perusteella valitaan muuttujaan liittyvät testitapaukset [ZHM97]. Tietovuomenetelmien ongelmana on kuitenkin niiden kalleus ja käytön hankaluus. Koska tarkastelun kohteena on vain yksi muuttuja kerrallaan, tietovuokaavion laatimisesta, sopivien testitapausten generoinnista ja tulosten analysoinnista muodostuva kustannus löydettyä

virhettä kohti muodostuu liiketaloudelliselta kannalta katsoen kohtuuttoman suureksi. Arvoriippuvien virheiden etsimiseen käytetäänkin liiketoimintaympäristöissä yleensä mustalaatikkotekniikoita [Bei95].

5.1.3 Mutaatiotestaus

Testitapausten riittävyyttä voidaan mitata myös mutaatiotestauksella, joka on virheiden istuttamiseen perustuva testaustekniikka. Mutaatiot ovat testattavasta ohjelmasta luotuja variantteja, joihin on tarkoituksellisesti lisätty virheitä ja joiden avulla arvioidaan testitapausten joukon kykyä löytää virheitä alkuperäisestä ohjelmasta [HoB89]. Mutaatio luodaan soveltamalla aitoja ohjelmavirheitä simuloivia mutaatio-operaattoreita alkuperäiseen ohjelmaan [LDJ99]. Testitapausten joukon tulisi erottaa alkuperäinen ohjelma ja mutantti toisistaan antamalla niiden testauksesta eriävä tulos, paitsi mikäli mutantti ja alkuperäinen ohjelma sattumalta ovat identtiset. Tämän erottelukyvyn perusteella testitapausten joukolle saadaan lasketuksi niin sanotut mutaatiopisteet (mutation score), jota voidaan käyttää mittarina testitapausten joukon riittävyydelle. Mutaatiotestauksen ongelmana ovat kuitenkin suuret kustannukset, jotka aiheutuvat mutaatioiden luomisesta, testitapausten soveltamisesta kaikkiin mutaatioihin sekä tulosten arvioinnista [DMM01].

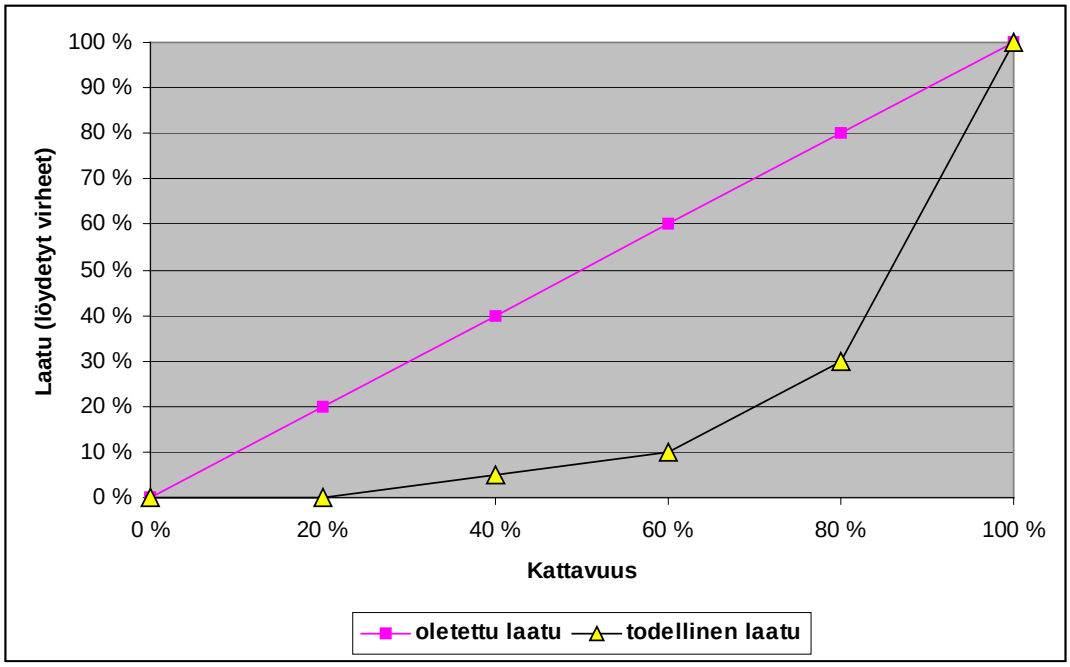
5.1.4 Muita rakenteellisen testauksen mittareita

Mainittujen, ohjelman kontorolli- tai tietovuon testaukseen perustuvien kattavuuksien lisäksi on olemassa useita muita kattavuuden mittoja [Kan96]. Kattavuuksia voidaan laskea muun muassa ohjelman tai yksikön tilojen tai silmukoiden [Bei90] läpikäynnin perusteella, poikkeuksille [SiH99], integrointitestauksessa integroitavien yksiköiden välisten kytkentöjen perusteella [OAA00], oliokielten erityisominaisuuksille [McM94] sekä useille muille ohjelman sekä rakenteellisille että myös toiminnallisille ominaisuuksille.

5.1.5 Kattavuuden suhde ohjelmakoodin laatuun

Testitapausten joukon kattavuutta voidaan käyttää myös sen laadun arviointiin [Mar91]. Mitä korkeampi kattavuusprosentti testitapausten joukolla saavutetaan, sitä enemmän ohjelmakoodissa olevia virheitä löydetään. Lukujen suhde ei kuitenkaan

tutkijoiden Williams, Mercer, Mucha ja Kapur [WMM01] mukaan ole lineaarinen, kuten yleisesti uskotaan, vaan eksponentiaalinen (Kuva 11). Tutkijat kuitenkin korostavat, että 100 %:n lausekattavuuden, tai minkä muun tahansa kattavuuden, saavuttaminen ei merkitse sitä, että ohjelmakoodi on täysin virheetön. Syynä tähän ovat polkutestaukseen liittyvät rajoitukset, joita käsittelen tarkemmin seuraavassa luvussa.



Kuva 11 **Ohjelmakoodista löydettyjen virheiden (=laatu) suhde kattavuuteen. Laatuasteikko kertoo, kuinka suuri osa ohjelman kaikista virheistä on löydetty [WMM01].**

Mikäli testitapausten joukko ei saavuta tavoiteltua kattavuutta, sitä tulee täydentää. Testitapausten täydentäminen jälkikäteen on kuitenkin kallista. Testaajan aikaa kuluu sen selvittämiseen, miksi tavoiteltua kattavuutta ei saavutettu ja mitkä ohjelman haarat tulee vielä testata. Myös testauksen kustannuksia ja kestoa on vaikea arvioida etukäteen mikäli saavutetut kattavuusluvut vaihtelevat suuresti projektista toiseen. Tämä taas vaikuttaa aikataulujen pitävyyteen ja sitä kautta kustannuksiin. Marick [Mar91] kehottaakin tilanteessa, jossa testitapausten joukolla saavutetaan matala kattavuusarvo, kiinnittämään huomiota myös testausprosessin parantamiseen. Testitapausten generointitekniikkaa voi olla tarpeen parantaa, testaaja voi tarvita lisäkoulutusta tai kyseisen sovellusalueeseen liittyvät testausongelmat voivat vaatia toimenpiteitä.

5.1.6 Polkutestauksen rajoitukset

Mikään rakenteellinen testaus, edes täydellinen polkukattavuus, ei yksinään takaa että ohjelmaa on testattu riittävästi, vaan aina tarvitaan myös toiminnallisuuden testausta [FeP96, DRM96, HLL94]. Polkutestauksella voidaan ainoastaan osoittaa, että ohjelmakoodissa olevat rivit on testattu. Polkutestauksella ei havaita virheellisiä tai puuttuvia funktioita [Bei84] eikä rajapinta-, tietokanta- tai alustusvirheitä [Hum89]. Ohjelman suorituspolkujen tietäminen ei myöskään kerro, miten luoda testitapauksia, joilla nämä polut suoritetaan. Tutkielmassa esitellyistä kattavuuksista täydellistä polkukattavuutta lukuun ottamatta ohjelmakoodissa olevat silmukat käydään läpi vain kerran, jolloin silmukan muihin iteraatiokierroksiin kätkeytyvät virheet jäävät löytymättä [Mar91].

Todellisessa ohjelmatuotannossa kattavuuksien laskemiseen tarvitaan aina automatiikkaa eli kattavuuden analysaattori. Analysaattori jäsentää ohjelmakoodin lisäämällä sinne omaa jäljityskoodia, jonka avulla suoritettujen ohjelmapolkujen seuraaminen ja kattavuuksien laskeminen mahdollistuu [Bin00]. Kattavuuden laskemisen mahdollistumisen lisäksi automatiikka auttaa vähentämään samanlaisia testitapauksia.

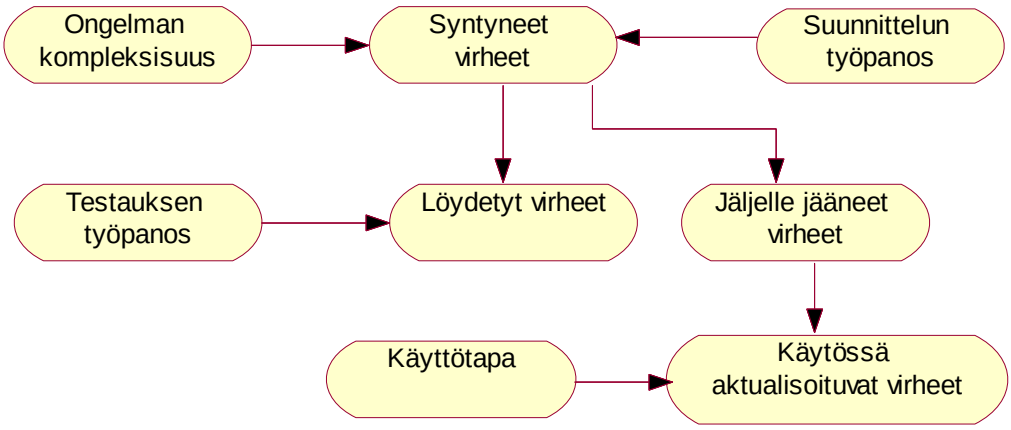
Tarvittavan testausautomatiikan vuoksi kattavuuksien laskeminen ja testausstrategioiden määrittely kattavuustavoitteiden perusteella ei ole yrityksen ensimmäisiä toimia yksikkö- ja integrointitestauksen määrittelyssä. Pienten, ohjelman toiminnan kannalta kriittisten, yksiköiden testauksessa niitä voidaan kuitenkin tarvittaessa pienimuotoisesti soveltaa manuaalisesti.

5.2 Toiminnallisen testauksen mittarit

5.2.1 Virheiden lukumäärä mittarina

Koska ohjelman virheet ovat sekä testaukseen että ohjelman käyttöön keskeisesti kuuluva tekijä, on luonnollista, että ne muodostavat yhden prosessin avainmittareista. Virheiden kappalemäärien laskemiseen perustuvat mittarit sisältävät kuitenkin monia mahdollisuuksia virhetulkintaan. Näistä tulkinnallisista riskeistä on tärkeää olla tietoinen, jotta tuotettujen lukujen perusteella ei tehtäisi virheellisiä johtopäätöksiä.

Eräs keskeinen ongelma virheiden määrässä mittarina on, ettei varmuudella voida sanoa, mittaako suure tuotteen vai prosessin ominaisuuksia. Testauksessa löydetty suuri virhemäärä voi kertoa joko kehnosta tuotteesta tai laadukkaasta testauksesta. Vastaavasti pieni virhemäärä ei automaattisesti tarkoita, että tuote on laadukas. Siksi on tärkeää tuntea myös luvun syntyyn vaikuttavia tekijöitä (Kuva 12).



Kuva 12 Virheiden syntyyn ja löytymiseen vaikuttavia tekijöitä [FeN00]

Ohjelmakoodiin syntyneiden virheiden määrään vaikuttaa sekä ohjelman ratkaiseman ongelman monimutkaisuus että ohjelman suunnitteluun käytetyt resurssit. Ohjelmakoodi sisältää todennäköisesti vähemmän virheitä, kun ohjelman suunnittelu tehdään huolella, siihen varataan riittävästi aikaa ja sen suorittavat tehtävään pätevät henkilöt. Ohjelmalle suoritetun testauksen määrä ja laatu puolestaan vaikuttavat siihen, kuinka suuri osa ohjelman virheistä löytyy ennen ohjelman julkistamista. Ohjelmaan julkistamisen jälkeen jääneiden virheiden löytyminen taas riippuu siitä, millä tavoin käyttäjät ohjelmaa käyttävät. Vähäinen löydettyjen virheiden määrä voi kertoa joko tuotteesta, jossa on vähän virheitä, tai tuotteesta, jota todellisuudessa käytetään erittäin vähän [FeN00]. Näistä seikoista johtuen pelkän virheiden määrän perusteella ei tule tehdä kovin pitkälle meneviä johtopäätöksiä tuotteesta tai prosessissa ennen tarkempien tietojen hankkimista kaikista asiaan vaikuttavista seikoista.

5.2.2 Mikä on virhe

Määrittelin tässä tutkielmassa käytettävän virhe-käsitteen sisällön luvussa 3.1 Vika – virhe – häiriö. Käsitteen sisältö on kuitenkin käytännössä usein hyvin hankala määrittää. Usein tulkinnan tekijästä riippuu se, mitä pidetään virheestä johtuvana häiriönä

ohjelman toiminnassa. Chillarege korostaa käyttäjän näkökulmaa ja päätyy määrittelemään häiriön tilanteeksi, jossa "asiakkaan odotukset eivät täyty ja/tai asiakas ei kykene tekemään tuotteella hyödyllistä työtä" [Chi96]. Asiakasnäkökulma on tärkeä liiketoiminnan lainalaisuuksille perustuvassa ohjelmistotuotannossa. Määritelmä myös kuvastaa hyvin sitä ongelmaa, joka liittyy virheen ja siitä mahdollisesti seuraavan ohjelman käyttäjälle näkyvän häiriön luokitteluun. Onko ohjelmassa ”virhe”, mikäli asiakkaan odotukset ovat teknisesti epärealistisia? Ovatko asiakkaan määritykset ohjelmatuotteelle olleet kenties hieman ylimalkaisia, jolloin ohjelmassa käyttäjän mielestä oleva ”virhe” onkin pelkästään seuraus asiakkaan implisiittisestä oletuksesta ohjelman oikeasta toiminnasta? Jollekin ohjelman tuottamana vastauksena antaman liukuluvun tarkkuus kolmella desimaalilla on virhe, kun taas toiselle käyttäjälle tarkkuus riittää. Perusteelliset määritykset auttavat jossakin määrin vähentämään tällaisia erilaisiin tulkintoihin perustuvia ”virheitä”.

Myös käyttäjän tapa käyttää ohjelmaa vaikuttaa suuresti siihen, aktualisoituuko ohjelmakoodin virhe koskaan häiriöksi. Ohjelma voi sisältää paljonkin löytymättömiä virheitä ilman, että asiakkaan käsitys tuotteen laadusta kärsii. Toisaalta yksittäinen virhe jollakin sellaisella ohjelmakoodin alueella, jota suoritetaan säännöllisesti, voi kenties aiheetta luoda käyttäjälle kuvan heikkolaatuisesta tuotteesta.

Virheen ja häiriön välisen eron ja sen seurausten ymmärtäminen korostuu, kun tuotteen ja sovelluskehitysprosessin laatua aletaan mitata. Sovelluskehitysprosessin eri vaiheissa "virheiden" määrällä voidaan käsittää eri asioita. Ohjelmakoodin tekijöille ja muokkaajille virheiden määrä on ohjelmakoodista löydettyjen virheiden määrä, kun taas sovellusta ainoastaan testaaville ja sen sisäistä rakennetta tuntemattomille virheiden määrä tarkoittaakin todellisuudessa ohjelman häiriöiden määrää [ChY96, FeP96]. Ero on tärkeää ymmärtää, mikäli eri kehitysvaiheissa tuotettuja lukuja halutaan verrata keskenään.

5.2.3 Virhetiheys

Eräs keskeinen käytössä oleva mitta ohjelmakoodin laadulle on virhetiheys (defect density). Ohjelmakoodissa voidaan ajatella olevan kahdenlaisia virheitä: tunnettuja virheitä, jotka on löydetty, sekä piileviä virheitä, jotka ovat järjestelmässä mutta joita

ei vielä ole löydetty. Virhetiheys lasketaan jakamalla tunnettujen virheiden määrä ohjelman koolla [FeP96]. Ohjelman kokoa mitataan useimmiten joko ohjelman fyysisen pituudella tai (LOC, DSI) toiminnallisuuden (Albrechtin toimintopisteet (function points) tai COCOMO 2:n objektipisteet) tai kompleksisuuden (McCaben syklomaattinen kompleksisuus) määrällä. Virhetiheyden kaava on seuraava:

virhetiheys = tunnettujen virheiden lukumäärä/ohjelman koko.

Kirjallisuudessa virhetiheydestä esitetään monenlaisia lukuja. Hatton mainitsee tyypilliseksi virhetiheydeksi esimerkiksi NASAn ohjelmissa 5-6 virhettä tuhatta koodiriviä (KLOC) kohti, huolimatta NASAn käytössä olevista resursseista ja voimakkaasta pyrkimyksestä virheiden estämiseen [Hat97]. Vastaavan suuruisia lukuja raportoivat myös Compton & Withrow [CoW90]. Beizer mainitsee perusteellisen moduuli- ja yksikkötestauksen jälkeiseksi virhetiheydeksi 2.4 virhettä / KLOC [Bei90].

Virhetiheys tarjoaa hyödyllistä informaatiota, mutta ennen lukujen vertailua muiden tuottamiin lukuihin tulee ottaa huomioon useita tekijöitä. Ensinnäkin epäselvästä virheen käsitteestä aiheutuu ongelmia, joita käsittelin jo aiemmin. Jotta esimerkiksi testausprosessin eri vaiheiden aikana tuotetut virhetiheysluvut olisivat vertailukelpoisia, on tärkeää varmistaa, että kaikki osapuolet laskevat samaa asiaa. Yksimielisyyttä ei myöskään ole siitä, miten ohjelman kokoa voitaisiin laskea yhdenmukaisella ja vertailukelpoisella tavalla. Lopuksi, vaikka virhetiheyttä pidetäänkin tuotteen ominaisuutena, luku tuotetaan virheiden etsintäprosessissa. Luku saattaa toisinaan kertoa enemmän virheiden löytämisen ja raportoinnin laadusta prosessissa kuin itse tuotteen laadusta [FeP96].

Virhetiheyttä käytetään usein ohjelman luotettavuuden mittarina. Fenton ja Neil argumentoivat voimakkaasti tätä vastaan, sillä heidän mukaansa ohjelman käytössä esiintyviä häiriöitä ei voida suoraan johtaa ohjelmassa jäljellä olevien virheiden määrästä [FeN99]. Virheiden vakavuutta on vaikea arvioida ennakolta, ja mikä vielä tärkeämpää, käyttäjien erilaiset ohjelman käyttötavat saavat aikaan sen, että on vaikea ennakoida, mitkä sovelluksessa olevat virheet todella aktualisoituvat ohjelman käytön häiriöiksi. Tutkijat painottavat myös sitä, että virhetiheys ei mittaa laatua siinä mielessä kuin asiakas laadun kokee eikä siksi sovellu sellaisenaan yleiseksi laadun

mittariksi. Myös Beizer varoittaa pelkän virhetiheyden käytön aiheuttamasta vääristymästä ohjelman luotettavuutta arvioitaessa [Bei84]. Hänen mukaansa ohjelma, jossa on kymmenen virhettä, joista ei aiheudu merkittäviä seurauksia, on luotettavampi kuin ohjelma, jossa on yksi katastrofaalisia seurauksia aiheuttava virhe. On selvää, että pelkän virheiden määrän laskemisen sijaan myös virheiden vakavuus tulee ottaa huomioon.

Virhetiheyden käyttö testausvaiheen mittarina edellyttää siis sekä ohjelman koon että virheiden määrän mittausta. Ohjelman koon mittaaminen suoritetaan tarvittavalla automatiikalla. Virheiden määrää lasketaan yleensä virheiden kirjausjärjestelmään talletettujen virheiden perusteella. Mikäli yrityksellä tai projektilla ei aiemmin ole ollut tapana kirjata ohjelman tekijän suorittamassa yksikkö- tai integrointitestausvaiheessa löydettyjä virheitä järjestelmään, mittarin tuottaman tiedon hyötyä suhteessa toimintatavan muutoksen aiheuttamiin kustannuksiin on harkittava. Lisäksi on määriteltävä, missä vaiheessa testausta mittaus aloitetaan. Yksikkötestaus on usein käytännössä hyvin kiinteä osa varsinaista ohjelmakoodin tekoa, ja esimerkiksi Beizer suosittaakin ohjelman tekijän suorittamaa yksityistä epäformaalia testausta ennen varsinaista virallista yksikkötestausta [Bei84].

Virheiden kirjauksen ja luokittelun tulee olla mahdollisimman yksinkertaista ja nopeaa. Tekijän itsensä suorittama virheiden kirjaus on varsin altista kirjaustavan ja –tarkkuuden muokkaukselle sen perusteella, millaisia seurauksia kirjauksen tekijä odottaa tehdyillä kirjauksilla itselleen olevan. Tämän vuoksi vertailukelpoisten virhetietojen saamiseksi yrityksen on tärkeää tiedottaa selkeästi ja rehellisesti mittaustietojen käyttötavat. Mittaustietojen, lähinnä virhemäärien, kohdentaminen henkilöön sekä henkilön suorituksen arviointi näiden tietojen perusteella on ehdottomasti estettävä, koska näin saadaan varmuudella tuotetuksi juuri halutunlaisia lukuja testauksen varhaisvaiheista [KBP02]. Suoritetun testauksen määrällä on selvä vaikutus virhetiheyden suuruuteen [FeN99]. Mikäli hyvä suoritus merkitsee suurten virhemäärien löytymistä, tekijä saattaa jäädä testaaman tuottamaansa ohjelmaa suhteettoman pitkäksi aikaa, jolloin sen siirtyminen järjestelmätestivaiheeseen ammattitaitoisten testaajien testattavaksi lykkääntyy. Vaihtoehtoisesti tekijä saattaa keskittää voimavaransa virheilmoitusten tehokkaaseen tuottamiseen. Jos taas pienet virhemäärät ovat

suotavia, virheitä on helppo jättää kirjaamatta järjestelmään. Tällaisella manipuloidulla tiedolla ei ole arvoa mittarina, ja pahimmillaan ohjelman tekijän testaukseen ja virheiden kirjaukseen kuluva aika kasvaa suhteettoman suureksi tuottavuuden vastaavasti romahtaessa ja tuotettujen mittaustulosten ollessa manipuloituja ja siksi harhaanjohtavia.

5.2.4 Virheiden priorisointi testauksessa

Testausvaiheessa löydettyjen virheiden lukumäärä yksinään ei kerro testauksen tehokkuudesta tai tuotteen hyvästä laadusta. Se, miten vakavana virhettä pidetään, vaikuttaa virheen priorisointiin ja sitä kautta resurssien allokointiin sekä virheen löytämiseen että korjaamiseen. Testauksessa tärkeää on löytää ensisijaisesti vakavimmat virheet. Virheiden vakavuuden luokitteluun on olemassa useita malleja.

Beizer [Bei90] on kehitellyt mitan virheen tärkeyden laskemiseen:

tärkeys ($) = yleisyys(korjauskulut +asennuskulut+seurauskustannukset)*

Virheen tärkeys määrittyy neljän ominaisuuden perusteella, joita ovat virheen yleisyys, korjauskulut, asennuskulut sekä seuraukset. Eniten huomiota tulee kiinnittää yleisimmin esiintyviin virhetyyppeihin. Virheen korjauskulut koostuvat kahdesta osatekijästä: virheen löytämisen aiheuttamista kuluista sekä sen korjauskuluista. Nämä kulut suurenevat mitä myöhäisemmässä vaiheessa kehitysprosessia virheet löydetään. Korjauskuluihin vaikuttaa myös järjestelmän koko: saman virheen löytäminen ja korjaaminen suuremmasta järjestelmästä vie enemmän aikaa ja tulee siksi kalliimmaksi.

Asennuskulut määräytyvät sovelluksen asennusten määrän mukaan. Mitä useampaan laitteeseen ohjelma on asennettu, sitä suuremmat kulut aiheuttaa virheen korjausta seuraava ohjelman uudelleenjakelu ja –asennus. Yksittäisen virheen seuraukset puolestaan voivat vaihdella vähäisestä haitasta suuriin inhimillisiin ja/tai liiketaloudellisiin menetyksiin.

Myös Binder kehottaa huomioimaan ohjelman suoristuksessa tapahtuvan häiriön potentiaaliset seuraukset testauksen määrää ja resursointia arvioinnissa [Bin00]. Ka-

ner, Falk ja Nguyen [KFN99] puolestaan ottavat käytännönläheisemmän lähestymistavan testausresurssien kohdentamisen arviointiin. Heidän tarkastelukulmansa on lähinnä sovellustestaajan, mutta mielestäni myös aikaisemmissa testausvaiheissa on tärkeää priorisoida sekä testauksen kohdetta, määrää että virheiden korjauksen prioriteetteja.

Tutkijoiden mukaan ensin tulee etsiä todennäköisimpiä virheitä. Ohjelmistotuotannossa puhutaan usein niin sanotusta Pareto-periaatteesta, yleisäännöstä, jonka mukaan 80% virheistä löytyy 20%:sta ohjelmakomponentteja [SBB02]. Tämän perusteella ohjelman yksiköt, moduulit, luokat tai ohjelmakomponentit, joista aiemmin on löytynyt virheitä, sisältävät niitä suurella todennäköisyydellä vielä lisää. Myös virheiden korjauksen yhteydessä syntyy helposti uusia virheitä, eli nämä tunnistetut ohjelman ”ongelma-alueet” kannattaa jatkossakin testata huolella [DRM96]. Myös Harrison pitää erityisen tärkeänä testaus- ja verifiointiresurssien keskittämistä juuri näiden virhealttiiden moduulien testaukseen [Har88]. Erityisesti regressiotestauksessa kannattaa hyödyntää aiemmin hankittua tietoa niistä ohjelman yksiköistä, joissa on esiintynyt eniten virheitä [DRM96]. Toisaalta Fenton ja Neil kritisoivat vahvasti tätä vastaan väittäen, että moduleista, joista on löydetty eniten virheitä ennen ohjelman julkaisua, paljastuu vastaavasti julkaisun jälkeen selvästi muita vähemmän virheitä. Näkökantojen eroa he perustelevat sillä, että suuri löytyneiden virheiden määrä kertoo ainoastaan siitä, että kyseessä olevat moduulit on testattu huolella, eli virhemäärien ero moduulien välillä selittyy testauksen määrän erolla [FeN00].

Toisena Kaner, Falk ja Nguyen [KFN99] kehottavat etsimään näkyvimpiä virheitä, eli niitä, jotka käyttäjä huomaa ensimmäisenä. Testaus keskitetään ohjelman eniten käytetyille alueille sekä sellaisiin piirteisiin, joilla tämä tuote erottuu kilpailijoistaan tai joissa on asiakkaan kannalta kriittistä toiminnallisuutta. Näin varmistetaan ohjelman ydintoiminnallisuus.

Kolmas testattava alue ovat eniten käytetyt ohjelman osat, sillä näillä alueilla sijaitsevat virheet esiintyvät säännöllisesti. Käyttäjälle muodostuu nopeasti kuva epäluotettavasta tai huonolaatuisesta tuotteesta, mikäli näitä ei korjata.

Neljäntenä keskeisenä kohteena ovat ne virheet, jotka sijaitsevat vaikeimmin korjattavilla ohjelman alueilla. Näin varmistetaan ohjelman tekijälle mahdollisimman pitkä aika suorittaa näiden virheiden korjausta. Viimeisenä testauksen alueena tutkijat kehottavat testaamaan ne ohjelman alueet, jotka testin tekijä parhaiten ymmärtää.

Koska kaikkia sovelluksen osia ei ole järkevää testata samalla tavoin, yrityksessä on syytä sopia kriteereistä, joiden perusteella yksikkö- ja integrointitestauksen voimavarat suunnataan. Pyrkimyksenä tulisi olla löytää ensisijaisesti yrityksen ja käyttäjän kannalta seurauksiltaan vakavimmat virheet mahdollisimman nopeasti testauksen alkuvaiheessa ja vasta toissijaisena tavoitteena mahdollisimman suuren virhemäärän löytäminen sovitun testausstrategian puitteissa.

5.3 Prosessimittarit

Testausprosessin eri osien välistä tehokkuutta voidaan vertailla kussakin vaiheessa löydettyjen virheiden määrällä. Kun tietyssä vaiheessa löydettyjen virheiden määrä suhteutetaan kunkin virheen löytämisen kestoon ja kustannuksiin, saadaan myös kuva siitä, onko kyseinen testausprosessin osa kustannustehokas [FeP96]. Prosessin tehokkuuden lisäksi on tärkeää myös löytää mittari, jonka perusteella voidaan tehdä päätös tuotteen siirtämisestä seuraavaan testauksen vaiheeseen.

5.3.1 Virheenpoiston tehokkuus

Vaiheen virheenpoiston tehokkuus (defect removal efficiency, DRE) tietyssä ohjelmistoprosessin vaiheessa ilmaisee prosentuaalisesti sen, paljonko kaikista ohjelman elinkaaren aikana löytyneistä virheistä löytyi ja korjattiin kyseisessä vaiheessa, esimerkiksi testauksen eri vaiheiden aikana [Jon96]:

*virheenpoiston tehokkuus = (vaiheessa korjatut virheet/kaikki virheet)*100.*

Ohjelman virheiden kokonaismääränä voidaan käyttää ohjelman tuotantovaiheessa sekä ensimmäisen vuoden ajan ohjelman julkaisun jälkeen löytyneiden virheiden määrää [Jon96] tai se voidaan yrittää ennustaa erilaisin matemaattisin mallein muun muassa ohjelman koon [Aki71, Lip82], kompleksisuuden [Hal75], suunnitteludokumenttien [OhA96], oliokielten ominaisuuksien [ChK94] tai kattavuuksien [VeM94]

perusteella. Virheenpoiston tehokkuuden laskenta on yksinkertaista ja kaavaa on helppo soveltaa myös esimerkiksi sen analysointiin, miten tehokkaita ohjelmistokehityksen eri testausvaiheet ovat eri tyyppisten virheiden havaitsemisessa [Jon96].

Jones mainitsee Yhdysvaltalaisten yritysten keskiarvoksi virheenpoiston tehokkuudessa noin 85 %, johtavien yritysten, kuten AT&T, IBM, Motorola ja Hewlett-Packard, saavuttaessa jopa 99 % parhaiten onnistuneissa projekteissaan. Luvut ovat koko ohjelmistotuotantoprosessin virheenpoiston tehokkuus ajalta ennen tuotteen julkaisua. Parhaat tulokset saavutetaan yhdistämällä sekä systemaattiset virheiden estämistoimet että katselmoinnit ja testaus. Mikään näistä menetelmistä yksinään ei takaa koko prosessin korkeaa virheenpoiston tehokkuutta [Jon96].

Vertailtaessa eri testausvaiheiden välisiä eroja virheenpoiston tehokkuudessa on muistettava, että kukin testausvaihe keskittyy hieman erilaisten virheiden löytämiseen. Esimerkiksi yksiköiden väliseen tiedonsiirtoon liittyviä virheitä ei ole mahdollista tai tarkoituksenmukaista löytää yksikkötestausvaiheessa [CrJ02].

Virheenpoiston tehokkuuden arvioinnissa on myös huomioitava testausympäristön aiheuttamat rajoitteet testaajan mahdollisuuksille löytää virheitä [CrJ02]. Kaikki aidon tuotantoympäristön ulkopuolella tehty testaus on suoritettu simuloidussa ympäristössä. Simuloidun ympäristön etuna on mahdollisuus käyttää ympäristöä projektin aikaisessa vaiheessa. Simuloidussa ympäristössä voidaan myös käyttää erilaisia testaus- ja virheenjäljitysapuvälineitä. Simuloitujen ympäristöjen haittoina ovat erilaisten ajoitussuhteiden vääristymät. Esimerkiksi kilpailutilanteet eivät vastaa todellista tilannetta. Myöskään ohjelman suorituskyvystä ei saada varmaa tietoa. Lisäksi yksikkö- ja integrointitestausvaiheissa resurssien hallintaa hoitaa yleensä jokin alirutiini, esimerkiksi tynkä, eikä varsinainen resurssienhallintaohjelma [Bei84].

Virheenpoiston tehokkuus tarjoaa mittarin yrityksen ohjelmistotuotantoprosessin eri vaiheiden tehokkuudesta yleisesti virheiden löytämisessä ja korjaamisessa sekä testausvaiheiden keskinäisistä eroista eri virhetyyppien löytämisessä. Yksittäisen projektin ja sen onnistumisen kannalta luku ei kuitenkaan ole erityisen hyödyllinen. Projektin ja sen eri osien välinen virheenpoiston tehokkuus on saatavissa vasta projektin päättymisen jälkeen eikä korjaaviin toimenpiteisiin enää kyseisen projektin

kuluessa ole mahdollisuutta. Tulosta voidaan kuitenkin käyttää vastaavanlaisten projektien arviointiin tulevaisuudessa.

5.4 Mittareiden käyttö

Yrityksessä erilaiset virheiden määristä johdetut mittarit ovat yleensä periaatteessa varsin yksinkertaisia ottaa käyttöön. Yrityksellä on lähes aina jo valmiina jonkinasteinen virheiden raportointi-, tallennus- ja käsittelymenettely, joka soveltuvin osin voidaan haluttaessa laajentaa myös yksikkö- ja integrointitestaukseen. Muodollinen virhekirjausten teko varsinaiseen virhekirjausjärjestelmään voi kuitenkin aiheuttaa yksikkö- ja integrointitestausvaiheissa liian suuria kustannuksia saatuun hyötyyn nähden. Mikäli virheiden määrän selvittämistä näissä vaiheissa pidetään tärkeänä, voisi jonkinlainen kevyt menettely, vaikkapa yksinkertainen tukkimiehen kirjanpito ja virhemäärien raportointi sen perusteella, olla ainakin yksikkötestausvaiheessa selvästi käyttökelpoisempi. Rakenteellisen testauksen mittarit vaativat lähes aina automatiikkaa ja ne on siksi tarkoituksenmukaista ottaa käyttöön yksikkötestauksen määrällisessä arvioinnissa vasta myöhemmin.

Mittareista käyttökelpoisimpia ovat sellaiset, jotka antavat palautetta tuotteesta tai prosessista jo yksittäisen projektin kuluessa, jolloin korjaavat toimenpiteet vielä ovat mahdollisia. On myös tärkeää antaa mittareiden tuottamaa tietoa palautteena ohjelmankehitysprosessille. Näin sekä mahdollistetaan itse prosessin parantaminen [Voa99] että motivoidaan mittaritietojen tuottajat tarkkuuteen tietojen keräämisessä ja tallennuksessa.

Ohjelmistotuotannon toiminnallisuuden mittareista puhuttaessa on muistettava, että ne ovat aina epäsuoria luonnehdintoja ominaisuuksista, joita ei voida mitata. Siksi mittareiden tuottamia lukuja tulee arvioida kriittisesti. Tarpeen mukaan käytössä olevia mittareita muokataan tai kehitetään edelleen tai otetaan käyttöön uusia, paremmin tarpeita vastaavia mittareita. Käyttö- ja vertailukelpoisten mittareiden kehittäminen ja validointi on pitkäaikainen prosessi. Mittareiden käyttö myös muuttaa prosessin kulkua ja aiheuttaa kustannuksia. Nämä seikat on tärkeää ottaa huomioon tavoiteltaessa eri testausvaiheiden parempaa laatua, hallintaa ja hallittua kehitystä.

6 Yhteenveto

Ohjelmistotuotannon kannalta kriittisiä tekijöitä ovat kustannukset, aikataulu, henkilöstö, toiminnallisuus sekä laatu [You95]. Kunkin yrityksen tulee omista, asiakaskuntansa sekä toimialansa sanelemista kilpailuehdoista lähtien miettiä, mitkä ovat sen kannalta tärkeimmät tekijät. Toisinaan ohjelmiston nopea valmistuminen ja jakelu markkinoille ennen kilpailijoita on tärkeämpää kuin sen mahdollinen virheettömyys. Vastaavasti on tilanteita, jolloin yrityksen ei kannata riskeerata liiketoiminnallista imagoaan tuotteella, joka asiakkaan näkökulmasta on huonolaatuinen. Nämä tekijät sanelevat myös yksittäisen yrityksen ja projektin yksikkö- ja integrointitestauksen tavoitetason. Mitään yleispätevää sääntöä tai toimintatapaa ei voida suoraan antaa.

Riittämätön yksikkö- ja integrointitestaus aiheuttavat virheiden siirtymistä seuraaviin testausvaiheisiin ja lisäävät näin ohjelman tuotantokustannuksia. Toisaalta myös liian perusteellinen testaus näissä vaiheissa kasvattaa kustannuksia. Liian perusteellisella testauksella ohjelman valmistuminen hidastuu ilman, että sen luotettavuus välttämättä kasvaa. Luotettavuuden parantumisen ja kustannusten välinen suhde saattaa myös kasvaa liiketoiminnallisesti kannattamattomaksi.

Organisaation on pyrittävä löytämään yrityksen kannalta optimaalinen testausstrategia, jossa ohjelma saavuttaa yksikkö- ja integrointitestausvaiheissa tarkoituksenmukaisen luotettavuustason mahdollisimman kustannustehokkaasti. Yksikkö- ja integrointitestausvaiheissa suoritettavan testauksen määrä on määriteltävä tarkasti ja selkeästi. Määritellyn tavoitteen saavuttamista tulee edesauttaa sopivalla koulutuksella, ohjeistuksella, välineistöllä ja huomioimalla testauksen vaatima aika myös sovelluskehitysprojektien tavoiteaikatauluissa. Mikäli näin ei ole, testauksen varhaisvaiheissa suoritetun testauksen määrä ja laatu määrittyvät kulloistenkin olosuhteiden mukaan: kuka ohjelman on toteuttanut, miten laajan testausvastuun hän katsoo itsellään olevan sekä millaiset mahdollisuudet hänellä on varsinaisen kehitystyön lisäksi suorittaa testausta.

Ennen kehittämistoimenpiteiden aloittamista yrityksen tulee selvittää, mikä on tällä hetkellä vallitseva tilanne. Alkutilakartoituksen avulla selvitetään yrityksen yksikkö-

ja integrointitestauksen nykytilannetta sekä ongelmia, puutteita ja tarpeita. Lisäksi alkutilakartoituksella voidaan löytää jo nyt käytössä olevia toimivia menetelmiä, joiden käyttöä voidaan laajentaa koko yritykseen. Alkutilakartoitus, toteutettuna esimerkiksi haastatteluin, auttaa myös sitouttamaan henkilöstöä kehittämistoimenpiteisiin tarjoamalla heille mahdollisuuden vaikuttaa valittaviin menetelmiin ja mittareihin.

Nykytilanteen kartoituksen jälkeen yritys määrittelee yksikkö- ja integrointitestausvaiheissa toteutettavat tehtävät ja sovellettavat menetelmät, dokumentaation ja käytettävät mittarit. Yrityksen tarpeisiin soveltuva virheiden luokittelujärjestelmä otetaan käyttöön. Vaiheissa toteutettavan testauksen liiketoiminnalliset realiteetit huomioiva riittävyys päätetään. Tämän riittävyyden toteutuminen testausvaiheessa varmistetaan jollakin yksinkertaisella, vaiheeseen nähden sopivalla mittarilla tai muulla selkeällä päätöskriteerillä. Lisäksi tuotteen ja prosessin laatua ja kehitystä seurataan sopivilla ja kustannustehokkailla mittareilla. Mittareiden antaman tiedon perusteella varhaisten testausvaiheiden kehitystä jatketaan, ohjataan ja tehtyjen toimenpiteiden vaikutusta seurataan. Yrityksen toimintaympäristö ja prosessin sisäiset tekijät huolehtivat siitä, että yksikkö- ja integrointitestauksen kehittäminen on jatkuva prosessi.

Lähteet

Aki71 Akiyama, F., An example of software system debugging. Information Processing, 71, 1971, 353-379.

Bei84 Beizer, B., Software system testing and quality assurance. Van Norstrand Reinhold, NY, 1984.

Bei90 Beizer, B., Software testing techniques. Van Norstrand Reinhold, NY, 1990.

Bei95 Beizer, B., Black-box testing. John Wiley & Sons, Inc, NY, 1995.

Bin00 Binder, Robert V., Testing object-oriented systems: models, patters, and tools. Addison-Wesley, Reading, MA, USA. 2000.

BKS98 Bassin, K.A., Kratschmer, T., Santhanam, P., Evaluating software development objectively. IEEE Software, 15, 6, 1998, 66-74.

BMK02 Butcher, M., Munro, H., Kratschmer, T., Improving software testing via ODC: three case studies. Software Testing and Verification, 41, 1, 2002, 31-44.

BS7925-1 BS 7925-1 Glossary of software testing terms. British Computer Society. 1998.

CBC92 Chillarege, R., Bhandari, I.S., Char J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M, Orthogonal Defect Classification-A Concept for In-Process Measurements. IEEE Transactions on Software Engineering, 18, 11, 1992, 943-956.

ChB95 Chillarege, R., Bassin, K.A., Software triggers as a function of time – ODC on field faults. Fifth IFIP Working Conference on Dependable Computing for Critical Applications, 1995. http://www.chillarege.com/odc/articles/trig/trig.html

Chi94 Chillarege, R., ODC for process measurement, analysis and control. Fourth International Conference on Software Quality, McLean, VA, USA, Oct 3-5, 1994, 143-158.

Chi96 Chillarege, R., What is Software Failure?, IEEE Transactions on Reliability, 45, 3, 1996, 354-355.

ChK94 Chidamber, S.R., Kemerer, C.F., A metrics suite for object oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 20, 6, 1994, 476-493.

ChP02 Chillarege, R., Prasad, K.R., Test and development process retrospektive - a case study using ODC triggers. Proceedings of International Conference on Dependable Systems and Networks (DSN 2002), Bethesda, MD, USA, 23-26 June, 2002, 669-678.

ChY96 Chen, T.Y., Yu, Y.T., On the expected number of failures detected by subdomain testing and random testing. IEEE Transactions on Software Engineering, 22, 2, 1996, 109-119.

CKC91 Chillarege, R., Kao, W.-L., Condit, G., Defect type and its impact on the growth curve. Proceedings of the 13th International Conference on Software Engineering, Austin, TX, USA, May 13-17, 1991, 246-255.

CoW90 Compton, B.T., Withrow, C., Prediction and control of Ada software defects. Journal of Systems Software, 12, 3, 1990, 199-207.

CrJ02 Craig, R.D., Jaskiel, S.P., Systematic software testing. Artech House Publishers, Norwood, MA, 2002.

DMM01 Delamaro, M.E., Maldonado, J.C., Mathur, A.P. Interface Mutation: An Approach for Integration Testing. IEEE Transactions on Software Engineering, 27, 3, 2001, 228-247.

DRM96 Duncan, I., Robson, D., Munro, M., Defect detection in code. Computer Science Technical Report 2. University of Durham, 1996.

EiR96 Eickelmann, N., Richardson, D., What makes one software architecture more testable than another?, Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, San Francisco, 1996, 65-67.

FeP96 Fenton, N.E., Pfleeger S.L., Software metrics: a rigorous and practical approach. International Thompson Computer Press, 1996.

FeN99 Fenton, N.E., Neil, M., A critique of software defect prediction models. IEEE Transactions on Software Engineering, 25, 5, 1999, 675-689.

FeN00 Fenton, N.E., Neil, M., Software metrics: roadmap. ICSE 2000, 22nd International Conference on on Software Engineering, Future of Software Engineering Track, Limerick Ireland, June 4-11, 2000, 357-370.

GeH88 Gelperin, D., Hetzel, B., The growth of software testing. ACM, 31, 6, 1988, 687-695.

Gui83 Guideline for lifecycle validation, verification and testing of computer software. National Bureau of Standards Report NBS FIPS 101. Washington, D.C. 1983.

Hal75 Halstead, M.H., Elements of software science. Elsevier, North-Holland, 1975.

Har88 Harrison, W., Using software metrics to allocate testing resources. Journal of Management Information Systems 4, 4, 1988, 93-105.

Hat97 Hatton, L., Reexamining the Fault Density Component Size Connection. IEEE Software 14, 2, 1997, 89-97.

HLL94 Horgan, J.R., London, S., Lyu, M.R., Achieving software quality with testing coverage measures. IEEE Computer, 27, 9, 1994, 60-69.

HoB89 Hoffman, D., Brealey, C., Module test case generation. Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification, Key West, Florida, USA, December 13-15, 1989, 97-102.

How86 Howden, W.A., A functional approach to program testing and analysis. IEEE Transactions on Software Engineering, 12, 10, 1986, 997-1005.

Hum89 Humphrey, W.S., Managing the software process. Addison-Wesley, Reading, MA, USA, 1989.

IEE610.12 ANSI/IEEE standard 610.12-1990: IEEE standard glossary of software engineering terminology, 1990.

IEEE829 ANSI/IEEE standard 829-1998: IEEE standard for software test documentation, 1998.

IEEE1008 ANSI/IEEE standard 1008-1987: IEEE standard for software unit testing, 1987.

IEEE1012 ANSI/IEEE standard 1012-1998: IEEE standard for software verification and validation, 1998.

IEEE1044 ANSI/IEEE standard 1044-1993: IEEE standard for software anomalies, 1993.

IEEE1044.1 ANSI/IEEE standard 1044.1-1995: IEEE guide to classification for software anomalies, 1995.

JoE94 Jorgensen, P.C. , Erickson, C. , Object-Oriented Integration Testing, Communications of the ACM, 37, 9, 1994, 30-38.

Jon96 Jones, C., Software defect-removal efficiency. IEEE Computer, 29, 4, 1996, 94-95.

Kan96 Kaner, C., Software negligence and testing coverage, 1996. http://www.kaner.com/coverage.htm

Kan00 Kaner, C., Measurement of the extent of testing. Pacific Northwest Software Quality Conference, Portland, Oregon, 2000. http://www.kaner.com/pnsqc.html

KBP02 Kaner, C., Bach, J., Pettichord, B., Lessons learned in software testing: A context-driven approach. Wiley Computer Publishing, N.Y., 2002.

KFN99 Kaner, C., Falk, J., Nguyen, H.Q., Testing computer software. Wiley Computer Publishing, N.Y., 1999.

LDJ99 Le Traon, Y., Devaux, D. and Jézéquel, J.-M., Self-Testable Components: from Pragmatic Tests to Design-for-Testability Methodology. In proc. of the Technology of Object-Oriented Languages and Systems Conference (TOOLS-Europe 99), Nancy (France), June 1999, 96-107.

Lip82 Lipow, M., Number of faults per line of code. IEEE Transactions on Software Engineering, 8, 4, 1982, 437-439.

MaM83 Martin, J., McClure, C., Software maintenance: the problem and its solutions. Prentice Hall, Eaglewood Cliffs, NJ, 1983.

Mar91 Marick, B. Experience with the cost of different coverage goals for testing. Proceedings of th Ninth Pacific Northwest Software Quality Conference, Portland, Oregon October 1991, 156-175.

McM94 McDaniel, R., McGregor, J.D., Testing the polymorphic interactions between classes. Department of Computer Science Technical Report, Clemson University, 1994.

McS01 McGregor, J.D., Sykes, D.A., A practical guide to testing object-oriented software. Addison-Wesley, Boston, 2001.

Mey79 Meyers, G.J., The Art of Software Testing. John Wiley & Sons. New York, 1979.

OAA00 Offutt, A.J., Abdurazik, A., Alexander, R.T., An Analysis Tool for Coupling-Based Integration Testing. The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00), Tokyo Japan, 2000, 172-178.

OhA96 Ohlsson, N., Alberg, H., Predicting error-prone software modules in telephone switches. IEEE Transactions on Software Engineering, 22, 12, 1996, 886-894.

Pre00 Pressman, R.S., Software Engineering: A Practitioner's Approach. McGraw-Hill, New York, 2000.

SBB02 Shull, F., Basili, V., Boehm, B., Brown A.W., Costa, P., Lindvall. M., Port, D., Rus, I., Tesoriero, R., Zelkowitz, M., What have we learned about fighting defects. Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02), Ottawa, Canada, June 04 - 07, 2002, 249-258. .

SiH99 Sinha, S., Harrold, M.J., Criteria for Testing Exception-Handling Constructs in Java Programs. Proceedings of International Conference on Software Maintenance (ICSM'99), Oxford, England, UK, 30 August - 3 September, 1999, 265-274.

SuC91 Sullivan, M., Chillarege, R., Software defects and their impact on system availability - a study of field failures in operating systems.

Digest of Papers: The 21 st Int. Symp. Fault-Tolerant Computing, 1991, 2-9.

SuR91 Su, J., Ritter, P.R., Experience in testing the motif interface. IEEE Software, 8, 2, 1991, 26-33.

VeM94 Veevers, A., Marshall, A.C., A relationship between software coverage metrics and reliability. Journal of Software testing, Verification and Reliability, 4, 1994, 3-8.

Voa99 Voas, J., Software quality's eight greatest myths. IEEE Software, 16, 5, 1999, 118-120.

Wey94 Weyuker, E.J., Can We Measure Sofwtare Testing Effectiveness? Proceedings of the First International Software Metrics Symposium, Baltimore, May 21-22, 1993, 100-107.

WMM01 Williams, T.W., Mercer, M.R., Mucha, J.P., Kapur, R., Code Coverage, What Does It Mean in Terms of Quality? Proceedings of the 2001 Annual Reliability and Maintainability Symposium, Philadelphia, PA, January 22-25, 2001, 420-424

You95 Yourdon, E., When Good Enough Software Is Best. IEEE Software, May, 1995, 70-81.

ZHM97 Zhu, H., Hall, P.A.V., May, J.H.R., Software unit test coverage and adequacy. ACM Computer Surveys, 29, 4, 1997, 366-427.